CprE 381, Computer Organization and Assembly-Level Programming

Lab 2 Report

Student Name: Rafat Momin

Submit a typeset pdf version of this on Canvas by the due date. Refer to the highlighted language in the lab document for the context of the following questions.

[Part 1 (a)] Draw the interface description (i.e., the “symbol” or high-level blackbox) for the MIPS register file. Which ports do you think are necessary, and how wide (in bits) do they need to be?

[Part 1 (b)] Create an N-bit register using this flip-flop as your basis.

[Part 1 (c)] Waveform.

[Part 1 (d)] What type of decoder would be required by the MIPS register file and why?

[Part 1 (e)] Waveform.

[Part 1 (f)] In your write-up, describe and defend the design you intend on implementing for the next part.

[Part 1 (g)] Waveform.

[Part 1 (h)] Draw a (simplified) schematic (i.e., components within the high-level blackbox) for the MIPS register file, using the same top-level interface ports as in your solution describe above and using only the register, decoder, and mux VHDL components you have created.

[Part 1 (i)] Waveform.

[Part 2 (b)] Draw a symbol for this MIPS-like datapath.

[Part 2 (c)] Draw a schematic of the simplified MIPS processor datapath consisting only of the component described in part (a) and the register file from problem (1).

[Part 2 (d)] Include in your report waveform screenshots that demonstrate your properly functioning design. Annotate what the final register file state should be.

[Part 3 (a)] Read through the mem.vhd file, and based on your understanding of the VHDL implementation, provide a 2-3 sentence description of each of the individual ports (both generic and regular).

[Part 3 (c)] Waveforms.

[Part 4 (a)] What are the MIPS instructions that require some value to be sign extended? What are the MIPS instructions that require some value to be zero extended?

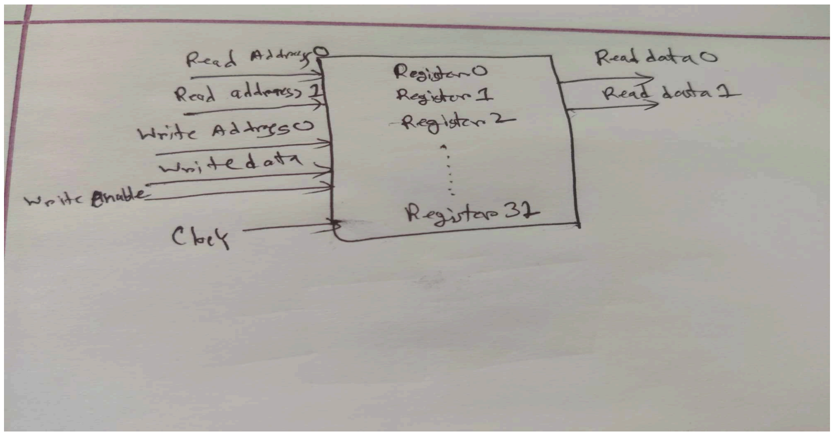
[Part 4 (b)] what are the different 16-bit to 32-bit “extender” components that would be required by a MIPS processor implementation?

[Part 4 (d)] Waveform.

[Part 5 (a)] what control signals will need to be added to the simple processor from part 2? How do these control signals correspond to the ports on the mem.vhd component analyzed in part 3?

[Part 5 (b)] Draw a schematic of a simplified MIPS processor consisting only of the base components used in part 2, the extender component described in part 4, and the data memory from part 3.

[Part 5 (c)] Waveform.



Part1a

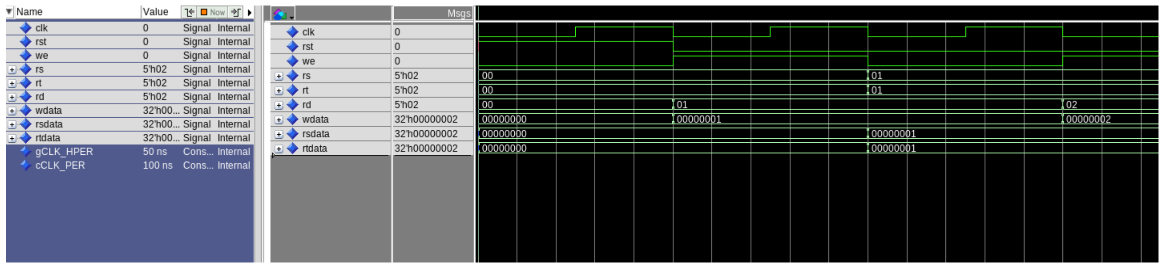
Required Ports and Their Functions:

Port Name	Direction	Bit-width	Function
`clk`	Input	1-bit	Clock signal for synchronous operation.
`reset`	Input	1-bit	Resets all registers to zero.
`regWrite`	Input	1-bit	Enables write operation when set to `1`.
`rs`	Input	5-bit	Read address for the first source register.
`rt`	Input	5-bit	Read address for the second source register.
`rd`	Input	5-bit	Write address for the destination register.
`writeData`	Input	32-bit	Data to be written into the register specified by `rd`.
`readData0`	Output	32-bit	Data read from the register specified by `rs`.
`readData1`	Output	32-bit	Data read from the register specified by `rt`.

VHDL Code (Register File Interface)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity RegFile is
   Port (
        clk        : in STD_LOGIC;                   -- Clock input for synchronous operation
        reset      : in STD_LOGIC;                   -- Active-high reset to initialize registers
        regWrite   : in STD_LOGIC;                   -- Write enable: active '1' when a write occurs
        rs         : in STD_LOGIC_VECTOR (4 downto 0);  -- 5-bit address for read port 1 (rs)
        rt         : in STD_LOGIC_VECTOR (4 downto 0);  -- 5-bit address for read port 2 (rt)
        rd         : in STD_LOGIC_VECTOR (4 downto 0);  -- 5-bit address for write port (rd)
        writeData  : in STD_LOGIC_VECTOR (31 downto 0); -- 32-bit data to write
        readData1  : out STD_LOGIC_VECTOR (31 downto 0); -- 32-bit output from read port 1
        readData2  : out STD_LOGIC_VECTOR (31 downto 0)  -- 32-bit output from read port 2
    );
end RegFile;
```



Name	Value		
clk	0	Signal	Internal
rst	0	Signal	Internal
we	0	Signal	Internal
rs	5'h02	Signal	Internal
rt	5'h02	Signal	Internal
rd	5'h02	Signal	Internal
wdata	32'h00...	Signal	Internal
rsdata	32'h00...	Signal	Internal
rtdata	32'h00...	Signal	Internal
gCLK_HPER	50 ns	Cons...	Internal
cCLK_PER	100 ns	Cons...	Internal

	Msgs
clk	0
rst	0
we	0
rs	5'h02
rt	5'h02
rd	5'h02
wdata	32'h00000002
rsdata	32'h00000002
rtdata	32'h00000002

Part1c(Hex)

Test Case Descriptions:

1. Reset the Register File:

- **Description:** Resets the register file, ensuring all registers are cleared to zero.

2. Write to Register 1:

- **Description:** Writes 0x00000001 to register 1.

3. Read from Register 1:

- **Description:** Reads value from register 1; expected 0x00000001 on rsdata and rtdata.

4. Write to Register 2:

- **Description:** Writes 0x00000002 to register 2.

5. Read from Register 2:

- **Description:** Reads value from register 2; expected 0x00000002 on rsdata and rtdata.

6. Write to and Read from Multiple Registers:

- **Description:** Writes to registers 4 and 8 (0x00000004 and 0x00000008 respectively) and then reads these values.

7. Reset Functionality:

- **Description:** Resets registers and ensures they are cleared.

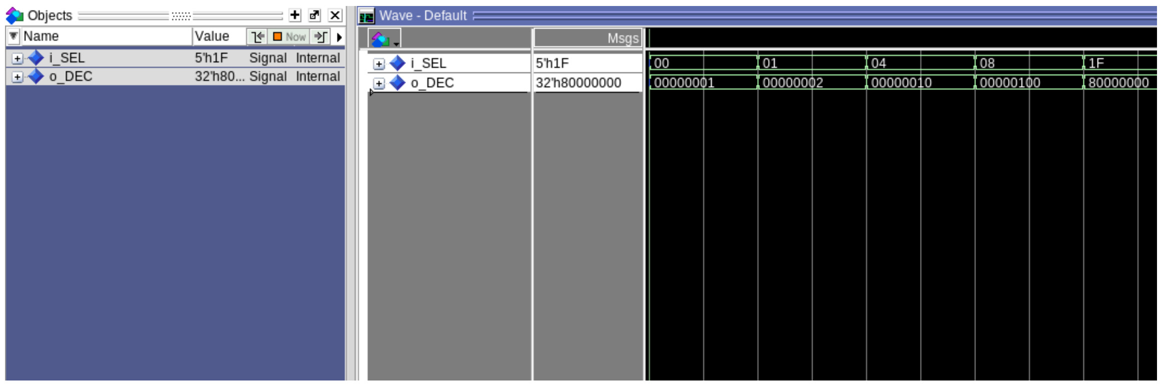
Part1d

For a MIPS register file, which contains 32 registers, a 5-to-32 decoder would be required. This is because the MIPS register file needs to select one of its 32 registers based on a 5-bit input.

Why a 5-to-32 Decoder:

- **N-bit Value:** The MIPS architecture uses 5 bits to specify a register (since 25=322^5 = 32).
- **2^N-bit Output:** The decoder will have a 32-bit output, with only one of those bits set to '1' at any given time, corresponding to the selected register.
- **Functionality:** When a specific 5-bit binary input is provided, the decoder will output a 32-bit value where only the bit corresponding to the input binary value is set to '1', while all other bits are '0'. This enables the correct register to be selected for read or write operations.

For example, if the input to the decoder is `00001`, the output will be `00000000000000000000000000000010`, selecting register 1.



```
Objects
Name      Value        
i_SEL     5'h1F        Signal  Internal
o_DEC     32'h80...    Signal  Internal

Wave - Default
                Msgs
i_SEL     5'h1F          00        01        04        08        1F
o_DEC     32'h80000000   00000001  00000002  00000010  00000100  80000000
```

Part 1e(Hex)
Overview:

- Verifies the decoder's functionality by applying specific inputs and observing corresponding outputs on the waveform.

Components:

- **i_SEL:** 5-bit input.
- **o_DEC:** 32-bit output.

Test Cases:

1. **00000 → 00000000000000000000000000000001:** First bit set.
2. **00001 → 00000000000000000000000000000010:** Second bit set.
3. **00100 → 00000000000000000000000000010000:** Fifth bit set.
4. **01000 → 00000000000000000000000100000000:** Ninth bit set.
5. **11111 → 10000000000000000000000000000000:** Last bit set.

Process:

1. **Initialize Signals:** Set `i_SEL` and `o_DEC`.
2. **Apply Test Cases:** Sequentially apply each test case.
3. **Observe Outputs:** Verify correct bit is set for each `i_SEL`.

Waveform Analysis:

- Confirms 5-bit input maps correctly to a unique 32-bit output with only one bit set to '1'.

Part1f

Design Description:

The 32-bit 32:1 multiplexer is designed to select one of the 32 input lines, each 32 bits wide, based on a 5-bit select input. The selected input is routed to a single 32-bit output.

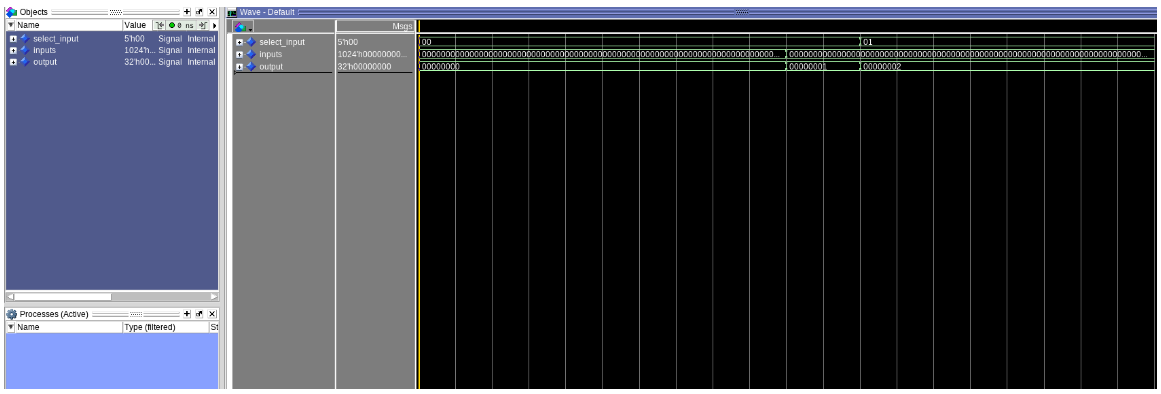
Design Approach:

The multiplexer is implemented using the behavioral approach in VHDL. This method defines the desired behavior of the circuit using a process block with a `case` statement to handle the selection logic.

Defense of the Design:

1. **Simplicity**: The behavioral approach using `case` statements makes the design straightforward, reducing complexity in the VHDL code.
2. **Efficiency**: Behavioral modeling is efficient for simulation, as it directly maps the selection logic without requiring detailed structural descriptions.
3. **Maintainability**: The code is easy to understand and maintain, making it simpler to troubleshoot and modify if necessary.
4. **Scalability**: This approach can be easily adapted or expanded to larger multiplexers if needed, maintaining a clean and organized structure.

By adopting the behavioral approach, the 32-bit 32:1 multiplexer design effectively balances simplicity, efficiency, and maintainability, making it a robust choice for the MIPS register file implementation.



Name	Value		
select_input	5'h00	Signal	Internal
inputs	1024'h...	Signal	Internal
output	32'h00...	Signal	Internal

```
select_input   5'h00             00                                          01
inputs         1024'h00000000... 0000000000000000000000000000000000000000... 000000000000000000000000000000000000000000000...
output         32'h00000000      00000000                        00000001    00000002
```

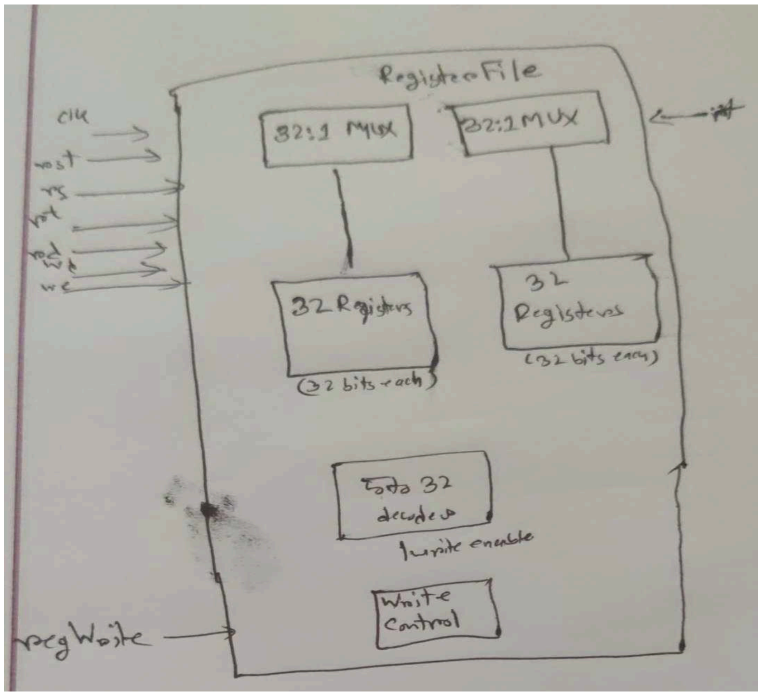
Part1g

Initialization:

  - The testbench initializes all inputs (d0 to d31) with distinct values to verify the multiplexer’s operation.
  - The `select_input` starts at "00000", selecting `d0`.

2. Selection and Output:
    - As `select_input` changes from "00000" to "00001", "00010", etc., the corresponding inputs (d0, d1, d2, etc.) are routed to the output.
    - The waveform shows the correct output for each selection. For example:
        - When `select_input` is "00000", the output is `d0`.
        - When `select_input` is "00001", the output is `d1`, and so on.
    - Each change in `select_input` results in the expected output value, confirming the correct operation of the multiplexer.
3. Final Verification:
    - The waveform verifies that the multiplexer successfully routes the correct 32-bit input to the output based on the 5-bit select input.
    - All 32 inputs are tested, demonstrating that the design works as intended.

Make sure to include a visual annotation on your screenshot to highlight the transitions and verify each input-output mapping.



Part1h

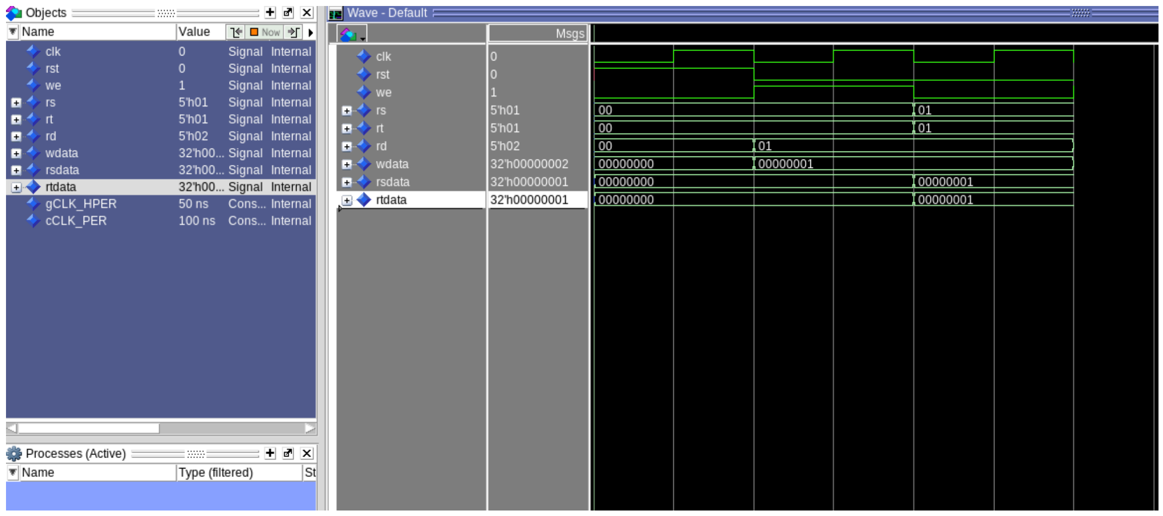
Overview

The MIPS register file consists of 32 registers, each of which is 32 bits wide. It includes:

- Two read ports (`rs`, `rt`): These allow simultaneous reading of two registers.
- One write port (`rd`): This is used to write data to a specific register.
- A control signal (`regWrite`): Determines whether a write operation occurs.
- A decoder: Selects which register to write into.
- Two 32:1 multiplexers: Select which registers to read.

Simplified Schematic Components

1. Registers (32 x 32-bit)
   - Each register stores 32 bits of data.
   - Register `$zero` (register 0) is hardwired to `0`.
2. Decoder (5-to-32)
   - Takes a 5-bit `rd` signal and enables only the corresponding register for writing.
3. Multiplexers (32:1 MUXs for `rs` and `rt`)
   - Selects one of the 32 registers based on `rs` and `rt` addresses.
4. Control Logic
   - Ensures writing happens only when `regWrite` is high.
   - Prevents writing to register `$zero`.



Objects

Name	Value		
clk	0	Signal	Internal
rst	0	Signal	Internal
we	1	Signal	Internal
rs	5'h01	Signal	Internal
rt	5'h01	Signal	Internal
rd	5'h02	Signal	Internal
wdata	32'h00...	Signal	Internal
rsdata	32'h00...	Signal	Internal
rtdata	32'h00...	Signal	Internal
gCLK_HPER	50 ns	Cons...	Internal
cCLK_PER	100 ns	Cons...	Internal

Wave - Default

	Msgs
clk	0
rst	0
we	1
rs	5'h01
rt	5'h01
rd	5'h02
wdata	32'h00000002
rsdata	32'h00000001
rtdata	32'h00000001

Part1i

The testbench file `tb_RegFile.vhd` is designed to simulate and verify the functionality of the MIPS register file by performing a series of write and read operations. Here's a breakdown of the key parts of the testbench:

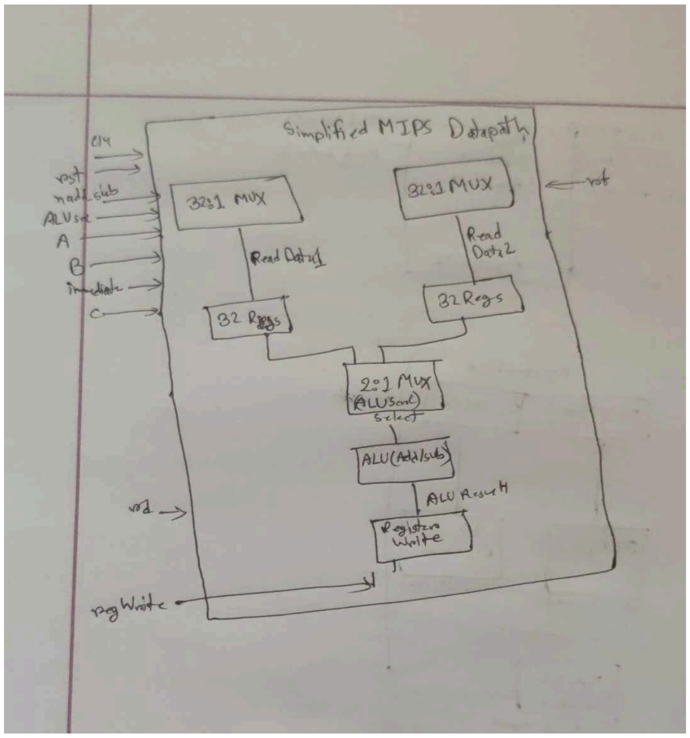
1. **Clock Generation**:
   - A clock signal (`clk`) is generated with a period of 10 ns. This signal drives the timing for the entire register file operation.
2. **Initialization and Reset**:
   - The reset signal (`rst`) is initially asserted (set to '1') for 20 ns to initialize the register file. After that, it is de-asserted (set to '0') to begin normal operation.
3. **Write Operations**:
   - The testbench enables writing (`write_enable` set to '1') and writes the value `X"AAAAAAAA"` to register 1 (`write_addr = "00001"`) and the value `X"BBBBBBBB"` to register 2 (`write_addr = "00010"`).
   - Each write operation takes place on the rising edge of the clock.
4. **Read Operations**:
   - After the write operations, the testbench performs read operations to verify the written values.
   - It reads from register 1 (`read_addr1 = "00001"`) and register 2 (`read_addr2 = "00010"`).
5. **Assertions**:
   - The testbench includes assertions to check if the read data matches the expected values. For example:

        - `assert (read_data1 = X"AAAAAAAA") report "Test failed for read address 1" severity error;`
        - `assert (read_data2 = X"BBBBBBBB") report "Test failed for read address 2" severity error;`
    - If the read data does not match the expected values, an error is reported.

Waveform Annotation:

When you annotate your waveform, consider highlighting the following key points:

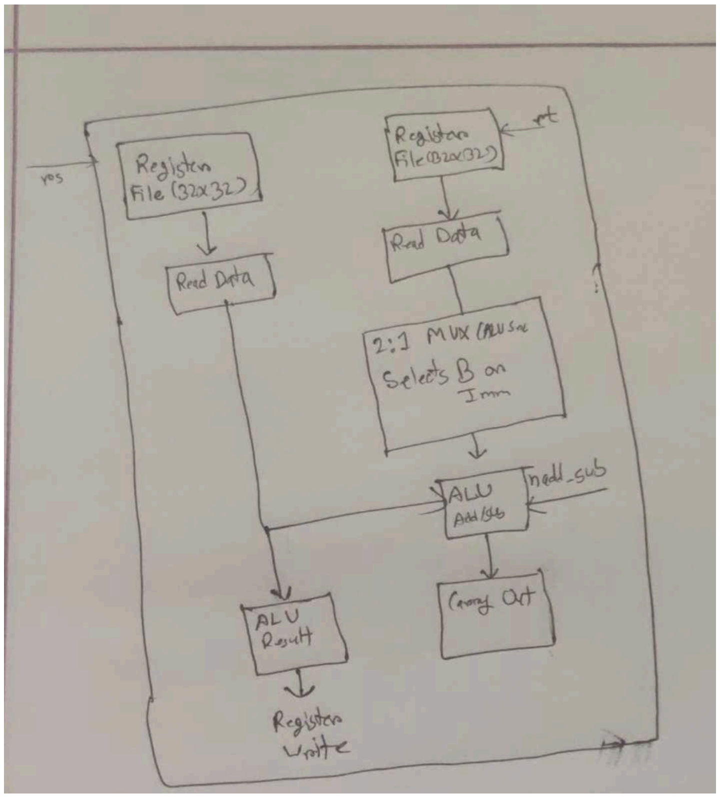
1. **Clock Signal**: Show the continuous clock signal with a period of 10 ns.
2. **Reset Signal**: Highlight the reset signal (`rst`) being asserted (set to '1') initially and then de-asserted (set to '0').
3. **Write Operations**:
    - Mark the time when `write_enable` is set to '1'.
    - Highlight the `write_addr` and `write_data` signals for the write operations to registers 1 and 2.
    - Indicate the rising edge of the clock where the write operation is performed.
4. **Read Operations**:
    - Mark the time when `write_enable` is set to '0'.
    - Highlight the `read_addr1`, `read_addr2`, and the corresponding `read_data1`, `read_data2` signals for the read operations.
    - Indicate the expected values for the read data and show the comparison with the actual read data.
5. **Assertions**: Note any assertions in the waveform to verify that the read data matches the expected values.



Part2b

Key Design Elements

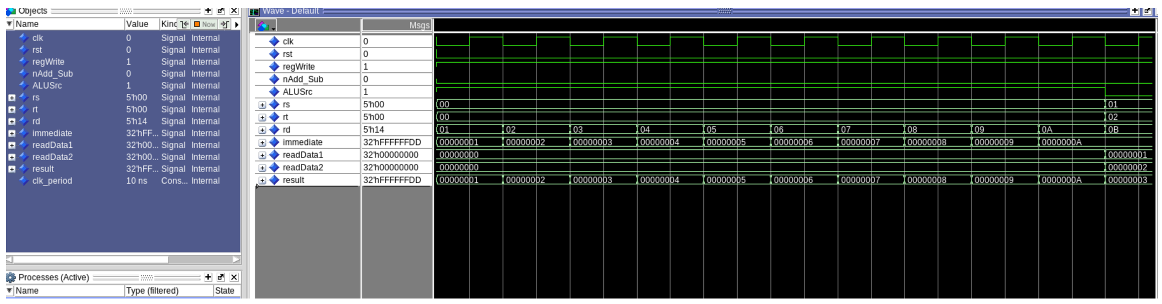
1. Register File:
   - Reads two registers (`rs`, `rt`).
   - Writes back to `rd` if `regWrite = 1`.
2. ALU:
   - Can perform addition or subtraction.
   - Uses a multiplexer to choose between register data or immediate (`ALUSrc`).
3. Multiplexer:
   - Selects between register `rt` or immediate based on `ALUSrc`.



Part2c

Components and Their Interactions

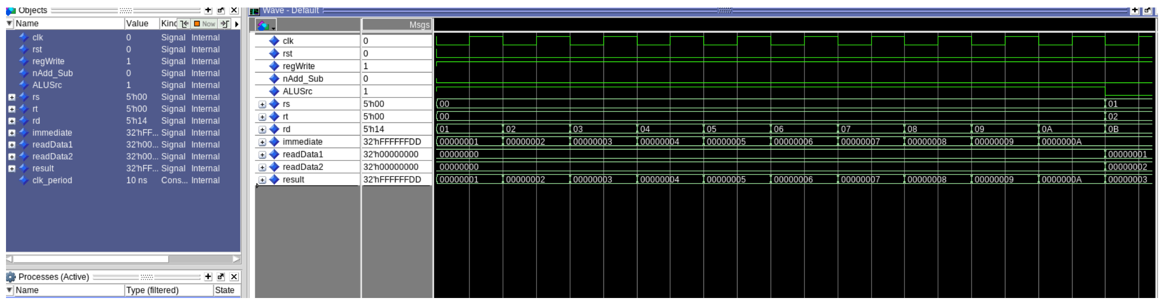
1. **Register File**:
   - Reads two registers (`rs` and `rt`).
   - Outputs their values (`read_data1` and `read_data2`).
2. **Multiplexer (2:1 MUX for ALU Src Selection)**:
   - Selects between `rt` value (`read_data2`) and an **immediate value**.
   - The selection is controlled by `ALUSrc`.
3. **ALU (Arithmetic Logic Unit)**:
   - Performs **addition** or **subtraction**.
   - Controlled by `nAdd_Sub` (`0` for add, `1` for subtract).
4. **Write-Back to Register File**:
   - After computation, the **result is written back** to the register file (`rd`).



```
Name        Value     Kind    Mode
clk         0         Signal  Internal
rst         0         Signal  Internal
regWrite    1         Signal  Internal
nAdd_Sub    0         Signal  Internal
ALUSrc      1         Signal  Internal
rs          5'h00     Signal  Internal
rt          5'h00     Signal  Internal
rd          5'h14     Signal  Internal
immediate   32'hFF... Signal  Internal
readData1   32'h00... Signal  Internal
readData2   32'h00... Signal  Internal
result      32'hFF... Signal  Internal
clk_period  10 ns     Cons... Internal
```

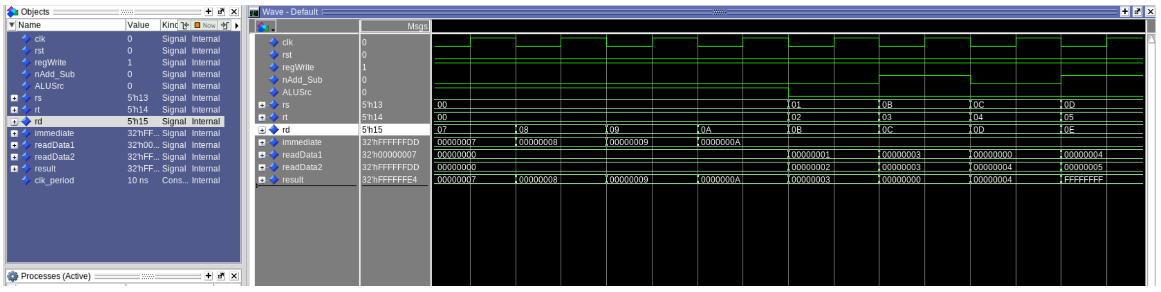
```
clk        0
rst        0
regWrite   1
nAdd_Sub   0
ALUSrc     1
rs         5'h00          00                                                                                    01
rt         5'h00          00                                                                                    02
rd         5'h14          01       02       03       04       05       06       07       08       09       0A       0B
immediate  32'hFFFFFFDD   00000001 00000002 00000003 00000004 00000005 00000006 00000007 00000008 00000009 0000000A
readData1  32'h00000000   00000000                                                                              00000001
readData2  32'h00000000   00000000                                                                              00000002
result     32'hFFFFFFDD   00000001 00000002 00000003 00000004 00000005 00000006 00000007 00000008 00000009 0000000A 00000003
```

Part2d



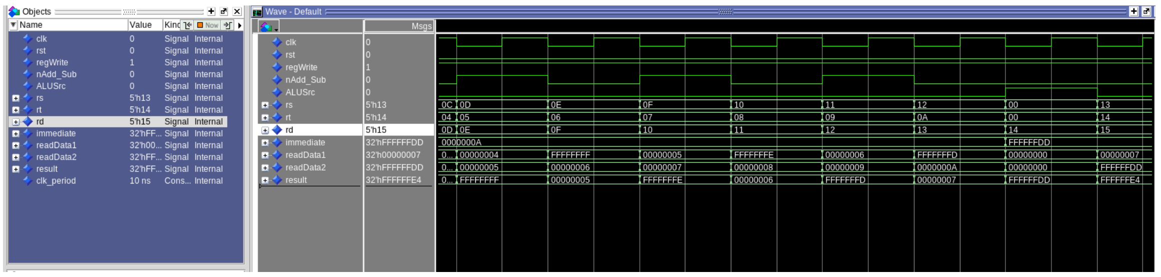
```
Name        Value     Kind    Mode
clk         0         Signal  Internal
rst         0         Signal  Internal
regWrite    1         Signal  Internal
nAdd_Sub    0         Signal  Internal
ALUSrc      1         Signal  Internal
rs          5'h00     Signal  Internal
rt          5'h00     Signal  Internal
rd          5'h14     Signal  Internal
immediate   32'hFF... Signal  Internal
readData1   32'h00... Signal  Internal
readData2   32'h00... Signal  Internal
result      32'hFF... Signal  Internal
clk_period  10 ns     Cons... Internal
```

```
clk        0
rst        0
regWrite   1
nAdd_Sub   0
ALUSrc     1
rs         5'h00          00                                                                                    01
rt         5'h00          00                                                                                    02
rd         5'h14          01       02       03       04       05       06       07       08       09       0A       0B
immediate  32'hFFFFFFDD   00000001 00000002 00000003 00000004 00000005 00000006 00000007 00000008 00000009 0000000A
readData1  32'h00000000   00000000                                                                              00000001
readData2  32'h00000000   00000000                                                                              00000002
result     32'hFFFFFFDD   00000001 00000002 00000003 00000004 00000005 00000006 00000007 00000008 00000009 0000000A 00000003
```



```
Name        Value     Kind    Mode
clk         0         Signal  Internal
rst         0         Signal  Internal
regWrite    1         Signal  Internal
nAdd_Sub    0         Signal  Internal
ALUSrc      0         Signal  Internal
rs          5'h13     Signal  Internal
rt          5'h14     Signal  Internal
rd          5'h15     Signal  Internal
immediate   32'hFF... Signal  Internal
readData1   32'h00... Signal  Internal
readData2   32'hFF... Signal  Internal
result      32'hFF... Signal  Internal
clk_period  10 ns     Cons... Internal
```

```
clk        0
rst        0
regWrite   1
nAdd_Sub   0
ALUSrc     0
rs         5'h13          00                                    01       0B       0C       0D
rt         5'h14          00                                    02       03       04       05
rd         5'h15          07       08       09       0A       0B       0C       0D       0E
immediate  32'hFFFFFFDD   00000007 00000008 00000009 0000000A
readData1  32'h00000007   00000000                              00000001 00000003 00000000 00000004
readData2  32'hFFFFFFDD   00000000                              00000002 00000003 00000004 00000005
result     32'hFFFFFFE4   00000007 00000008 00000009 0000000A 00000003 00000000 00000004 FFFFFFFF
```



```
Name        Value     Kind    Mode
clk         0         Signal  Internal
rst         0         Signal  Internal
regWrite    1         Signal  Internal
nAdd_Sub    0         Signal  Internal
ALUSrc      0         Signal  Internal
rs          5'h13     Signal  Internal
rt          5'h14     Signal  Internal
rd          5'h15     Signal  Internal
immediate   32'hFF... Signal  Internal
readData1   32'h00... Signal  Internal
readData2   32'hFF... Signal  Internal
result      32'hFF... Signal  Internal
clk_period  10 ns     Cons... Internal
```

```
clk        0
rst        0
regWrite   1
nAdd_Sub   0
ALUSrc     0
rs         5'h13          0C 0D       0E       0F       10       11       12       00       13
rt         5'h14          04 05       06       07       08       09       0A       00       14
rd         5'h15          0D 0E       0F       10       11       12       13       14       15
immediate  32'hFFFFFFDD   0000000A                                                  FFFFFFDD
readData1  32'h00000007   0... 00000004 FFFFFFFF 00000005 FFFFFFFE 00000006 FFFFFFFD 00000000 00000007
readData2  32'hFFFFFFDD   0... 00000005 00000006 00000007 00000008 00000009 0000000A 00000000 FFFFFFDD
result     32'hFFFFFFE4   0... FFFFFFFF 00000005 FFFFFFFE 00000006 FFFFFFFD 00000007 FFFFFFDD FFFFFFE4
```

Part2d

Test Case 1: addi $1, $0, 1

Description:

- This instruction adds the immediate value 1 to the value in register $0 (which is always 0) and stores the result in register $1.
- **Expected Output:** Register $1 should have the value 1.

Waveform Annotation:

- **clk**: Clock signal.
- **rst**: Reset signal, which is initially asserted to reset the system.
- **regWrite**: Set to '1' to enable writing to the register file.
- **nAdd_Sub**: Set to '0' for addition.
- **ALUSrc**: Set to '1' to select the immediate value as the second operand.
- **rs**: Register source $0.
- **rd**: Register destination $1.
- **immediate**: Immediate value 1.
- **result**: The result should be 1, written to register $1.

Test Case 2: addi $2, $0, 2

Description:

- This instruction adds the immediate value 2 to the value in register $0 and stores the result in register $2.
- **Expected Output:** Register $2 should have the value 2.

Waveform Annotation:

- Similar to Test Case 1, but with rd set to $2 and immediate set to 2.

Test Case 3: addi $3, $0, 3

Description:

- This instruction adds the immediate value 3 to the value in register $0 and stores the result in register $3.
- **Expected Output:** Register $3 should have the value 3.

Waveform Annotation:

- Similar to Test Case 1, but with rd set to $3 and immediate set to 3.

Test Case 4: add $11, $1, $2

Description:

- This instruction adds the value in register $1 to the value in register $2 and stores the result in register $11.
- **Expected Output:** Register $11 should have the value 3 (1 + 2).

Waveform Annotation:

- **clk**: Clock signal.
- **rst**: Reset signal.
- **regWrite**: Set to '1' to enable writing to the register file.
- **nAdd_Sub**: Set to '0' for addition.
- **ALUSrc**: Set to '0' to select the second register value as the second operand.
- **rs**: Register source $1.
- **rt**: Register source $2.
- **rd**: Register destination $11.
- **result**: The result should be 3, written to register $11.

Test Case 5: sub $12, $11, $3

Description:

- This instruction subtracts the value in register $3 from the value in register $11 and stores the result in register $12.
- **Expected Output:** Register $12 should have the value 0 (3 - 3).

Waveform Annotation:

- **clk**: Clock signal.
- **rst**: Reset signal.
- **regWrite**: Set to '1' to enable writing to the register file.
- **nAdd_Sub**: Set to '1' for subtraction.
- **ALUSrc**: Set to '0' to select the second register value as the second operand.
- **rs**: Register source $11.
- **rt**: Register source $3.
- **rd**: Register destination $12.
- **result**: The result should be 0, written to register $12.

Continue this pattern for each test case, describing the instruction, expected output, and relevant waveform annotations.

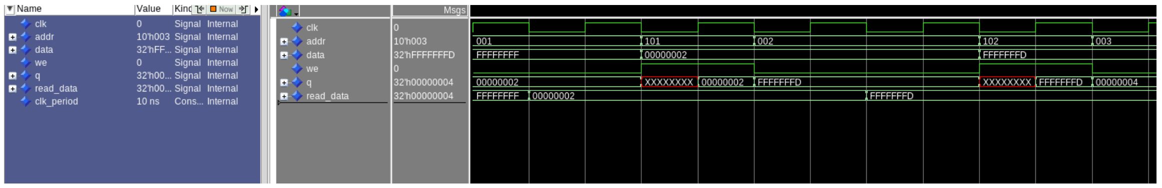
Part 3a: Description of Ports

Generic Ports:

1. **DATA_WIDTH**: This generic parameter specifies the width of the data bus. In this case, it is set to 32 bits.
2. **ADDR_WIDTH**: This generic parameter specifies the width of the address bus. In this case, it is set to 10 bits.

Regular Ports:

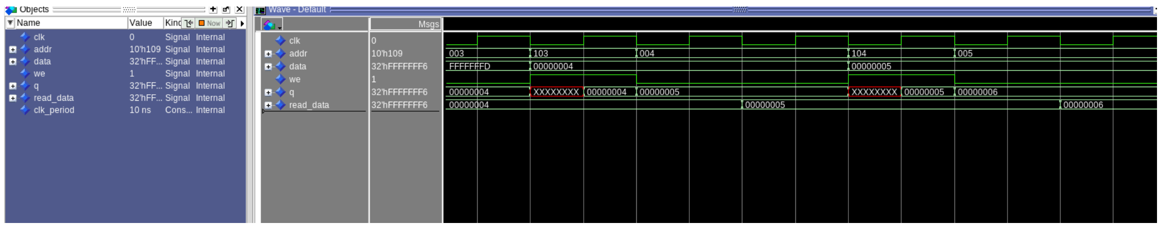
1. **clk**: This is the clock signal input. The memory operations (read/write) are synchronized with the rising edge of this clock.
2. **addr**: This is the address input for accessing the memory. It specifies which memory location to read from or write to.
3. **data**: This is the data input for writing to the memory. When a write operation is enabled (`we = '1'`), the data on this port is written to the memory location specified by the `addr` input.
4. **we**: This is the write enable signal. When this signal is '1', a write operation is performed. When it is '0', only read operations occur.
5. **q**: This is the data output port. It holds the data read from the memory location specified by the `addr` input. The output is valid after the rising edge of the clock when a read operation is performed.



Name	Value	Kind	
clk	0	Signal	Internal
addr	10'h003	Signal	Internal
data	32'hFF...	Signal	Internal
we	0	Signal	Internal
q	32'h00...	Signal	Internal
read_data	32'h00...	Signal	Internal
clk_period	10 ns	Cons...	Internal

	Msgs
clk	0
addr	10'h003
data	32'hFFFFFFFD
we	0
q	32'h00000004
read_data	32'h00000004

Part3c



Name	Value	Kind	
clk	0	Signal	Internal
addr	10'h109	Signal	Internal
data	32'hFF...	Signal	Internal
we	1	Signal	Internal
q	32'hFF...	Signal	Internal
read_data	32'hFF...	Signal	Internal
clk_period	10 ns	Cons...	Internal

	Msgs
clk	0
addr	10'h109
data	32'hFFFFFFF6
we	1
q	32'hFFFFFFF6
read_data	32'hFFFFFFF6

Part3c

Testbench Description

The testbench `tb_dmem.vhd` is designed to validate the functionality of the memory module (`mem.vhd`). It performs the following tasks:

1. **Clock Generation:** A 10ns period clock signal is generated to synchronize memory operations.
2. **Memory Initialization:** Reads pre-loaded values from `dmem.hex` file and ensures the memory is initialized correctly.
3. **Read Operations:** Reads the first 10 values from memory (starting at address `0x000`).
4. **Write Operations:** Writes the read values to new locations starting at `0x100`.
5. **Read-Back Verification:** Reads back the newly written values to ensure data integrity.
6. **Synchronization Issues Fixed:** Extra delay cycles were introduced to ensure correct data propagation and avoid `XXXXXXX` values.

Waveform Analysis

The waveform shows the **memory read and write operations** over multiple clock cycles. Key observations:

1. **Clock Signal (clk):** Alternates between `0` and `1`, ensuring synchronization.
2. **Address (addr):** Increments during read and write cycles.
3. **Data (data):** Shows values being written into memory.
4. **Write Enable (we):** Set to `1` during write operations, otherwise `0` for read.
5. **Read Data (q):** Initially shows undefined (`XXXXXXX`) but stabilizes once valid data is available.
6. **Successful Writes and Reads:** Values are correctly stored and retrieved, confirming memory functionality.

Conclusion

- The testbench successfully verifies memory behavior.
- Initial `XXXXXXX` values indicate **memory uninitialized state**, which is resolved once valid values are accessed.
- The waveform confirms correct read and write operations, validating the memory module implementation.

Part4a

Instructions Requiring Sign Extension

Certain MIPS instructions operate on **16-bit immediate values** but need to extend them to **32-bit** for proper execution. **Sign extension** is used when the immediate value needs to retain its signed nature (positive or negative). This is done by copying the **most significant bit (MSB)** of the 16-bit value into the upper 16 bits of the 32-bit representation.

MIPS Instructions that Require Sign Extension:

- **Arithmetic and Logical Immediate Instructions:**
  - `addi` (Add Immediate)
  - `slti` (Set Less Than Immediate)
  - `sltiu` (Set Less Than Immediate Unsigned) *(Despite "Unsigned" in its name, it still requires sign extension.)*
- **Load/Store Instructions:**
  - `lw` (Load Word)
  - `lh` (Load Halfword)
  - `lhu` (Load Halfword Unsigned) *(This instruction zero-extends, not sign-extends.)*
  - `lb` (Load Byte)
  - `lbu` (Load Byte Unsigned) *(This instruction zero-extends, not sign-extends.)*
  - `sw` (Store Word)
  - `sh` (Store Halfword)
  - `sb` (Store Byte)
- **Branch Instructions:**
  - `beq` (Branch if Equal)
  - `bne` (Branch if Not Equal)
  - `bgez` (Branch if Greater Than or Equal to Zero)
  - `bgtz` (Branch if Greater Than Zero)
  - `blez` (Branch if Less Than or Equal to Zero)
  - `bltz` (Branch if Less Than Zero)
  - **(All branch instructions use sign-extended 16-bit offsets for PC-relative addressing.)**

Instructions Requiring Zero Extension

Some instructions use **16-bit immediate values** that do not represent signed numbers but must be extended to **32-bit** for computation. **Zero extension** is used in these cases, where the upper 16 bits of the 32-bit representation are filled with **zeros** instead of replicating the MSB.

MIPS Instructions that Require Zero Extension:

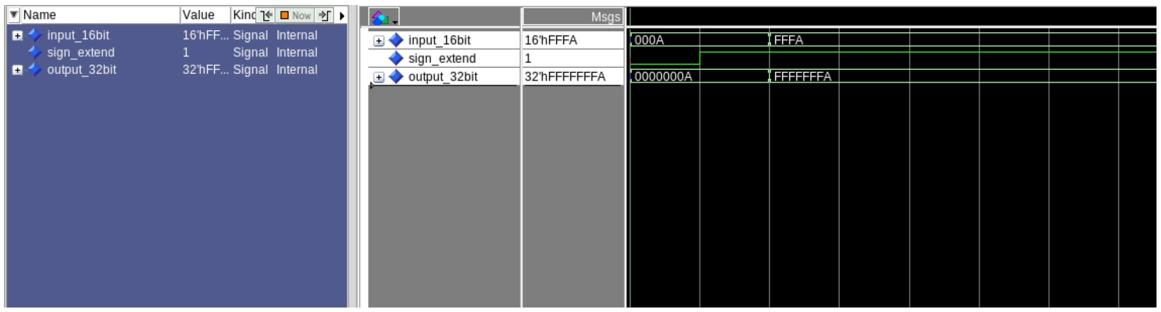
- **Bitwise Logical Operations with Immediate Values:**
  - `andi` (AND Immediate)
  - `ori` (OR Immediate)
  - `xori` (XOR Immediate)
- **Load Unsigned Instructions:**
  - `lbu` (Load Byte Unsigned)
  - `lhu` (Load Halfword Unsigned)

Part4b

To support **16-bit to 32-bit conversions**, a MIPS processor requires **two types of extenders**: a **sign extender** and a **zero extender**. These components ensure that immediate values are correctly formatted before being processed by the ALU or stored in registers.

1. **Sign Extender:**
    - Used for **signed** 16-bit values, where the most significant bit (MSB) is extended to maintain the sign.
    - Required by instructions such as **addi, slti, lw, lh, lb, sw, sh, sb, beq, bne, bgez, bgtz, blez, bltz**.
    - Ensures negative numbers remain negative and positive numbers remain positive when extended.
2. **Zero Extender:**
    - Used for **unsigned** 16-bit values, where the upper 16 bits are filled with zeros.
    - Required by instructions such as **andi, ori, xori, lbu, lhu**.
    - Ensures unsigned values remain positive when stored in 32-bit registers.
3. **Extender Multiplexer (ExtMux):**
    - Since MIPS uses **both** types of extenders, a multiplexer selects whether to apply **sign** or **zero** extension based on the instruction type.

Together, these extenders ensure that **all** 16-bit immediate values are correctly processed within the **32-bit** MIPS architecture.



Name	Value	Kind	
input_16bit	16'hFF...	Signal	Internal
sign_extend	1	Signal	Internal
output_32bit	32'hFF...	Signal	Internal

	Msgs		
input_16bit	16'hFFFA	000A	FFFA
sign_extend	1		
output_32bit	32'hFFFFFFFA	0000000A	FFFFFFFA

Part4d

The `Extender_tb` is the testbench that verifies the functionality of the `Extender` entity. The key elements are:

1. **Signal Declarations:**
    - `input_16bit`: Simulates the 16-bit input vector.
    - `sign_extend`: Simulates the control signal for choosing the extension method.
    - `output_32bit`: Observes the 32-bit output vector from the `Extender` component.
2. **Instantiation of `Extender`:**
    - The `Extender` component is instantiated within the testbench architecture.
    - Ports are mapped to the corresponding test signals.
3. **Process Block with Test Vectors:**
    - Provides stimulus to the `input_16bit` and `sign_extend` signals to test various cases.
    - **Test Case 1: Zero Extension** (e.g., input `0000 0000 0000 1010`, expected output `0000 0000 0000 0000 0000 0000 0000 1010`).
    - **Test Case 2: Sign Extension Positive Value** (e.g., input `0000 0000 0000 1010`, expected output `0000 0000 0000 0000 0000 0000 0000 1010`).
    - **Test Case 3: Sign Extension Negative Value** (e.g., input `1111 1111 1111 1010`, expected output `1111 1111 1111 1111 1111 1111 1111 1010`).

    - Wait statements ensure that each test case is applied sequentially.

Part5a

Control Signals for Load/Store Instructions: To incorporate load and store instructions into the MIPS-like datapath, additional control signals are required. The primary control signals are:

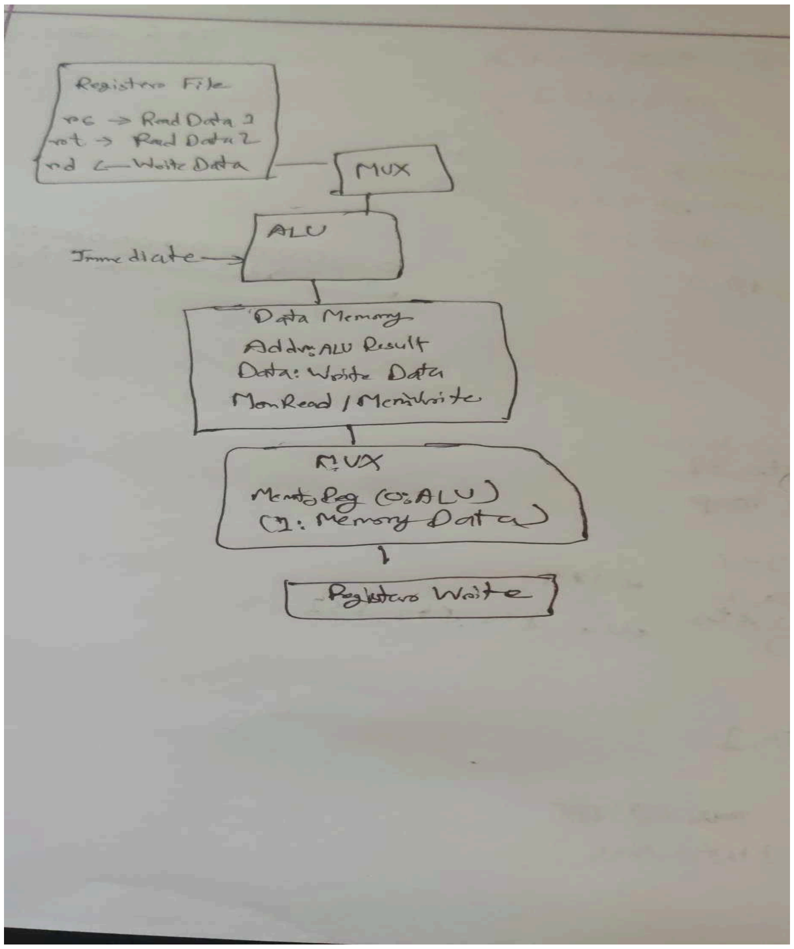
1. MemRead: Indicates if the instruction is a load instruction, requiring data to be read from memory.
2. MemWrite: Indicates if the instruction is a store instruction, requiring data to be written to memory.
3. MemToReg: Selects whether the data for the register write-back comes from memory (load) or the ALU.

Ports on mem.vhd component:

- Address: The memory address for load/store operations.
- WriteData: The data to be written into memory (used in store instructions).
- ReadData: The data read from memory (used in load instructions).
- MemRead: Control signal to enable memory read operations.
- MemWrite: Control signal to enable memory write operations.

These control signals correspond to the ports on the mem.vhd component and ensure proper data flow during load and store operations in the MIPS processor.

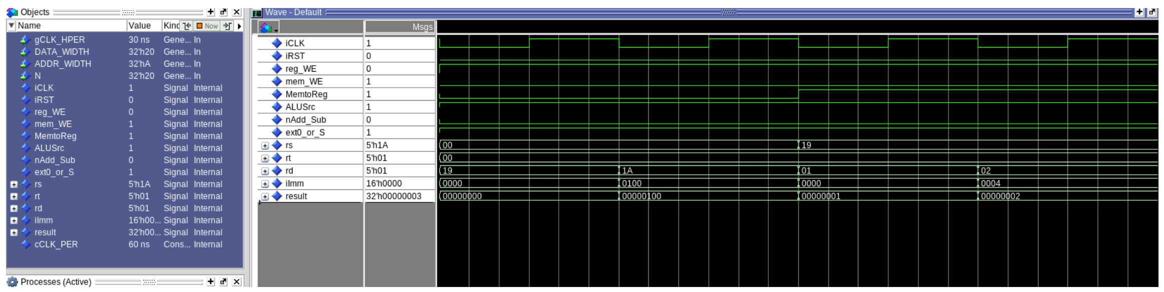
Summary: By integrating the MemRead, MemWrite, and MemToReg control signals, and connecting them to the appropriate ports on the mem.vhd component, we enable our MIPS processor to handle load and store instructions effectively. This enhancement ensures data can be properly read from or written to memory during program execution.



Part5b
The simplified MIPS processor datapath consists of the following signal flow:

1. Instruction Fetch Stage:
    - The instruction (assumed pre-decoded) provides:
        - Register Addresses (`rs`, `rt`, `rd`)
        - Opcode
        - Immediate Value (`imm16`)
2. Register Read & Immediate Extension:
    - `rs` and `rt` are used to read values from the Register File.
    - The Extender sign-extends `imm16` to a 32-bit value.
    - MUX before ALU selects between register value (`rt`) or sign-extended `imm16` (`ALUSrc` control signal).
3. ALU Operation:
    - Performs calculations like address computation for load (`lw`) and store (`sw`).
    - For `lw/sw`, the ALU computes memory address using `rs + imm16`.
4. Memory Access Stage (Load & Store):
    - Load (`lw`):
        - Memory address (computed by ALU) is used to fetch data.
        - MUX before Register File selects memory data (`MemtoReg = 1`).
    - Store (`sw`):
        - The register value (`rt`) is written into memory.

5. Writeback Stage (for `lw` only):
   - The MUX before Register File selects between:
     - ALU result (for arithmetic instructions).
     - Memory data (`MemtoReg = 1` for `lw`).

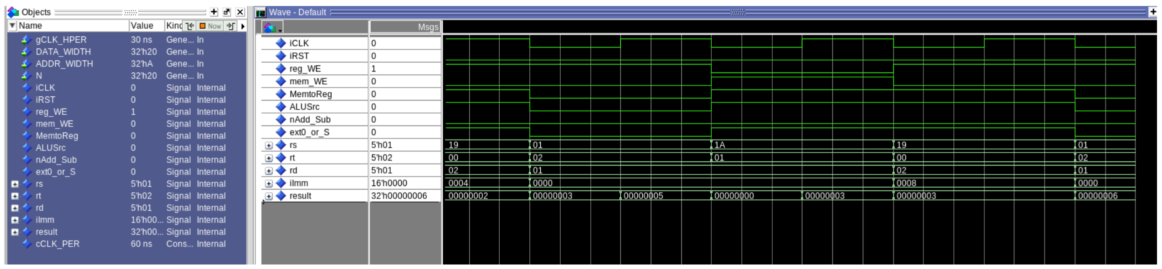


Objects

Name	Value	Kind	
gCLK_HPER	30 ns	Gene...	In
DATA_WIDTH	32'h20	Gene...	In
ADDR_WIDTH	32'hA	Gene...	In
N	32'h20	Gene...	In
iCLK	1	Signal	Internal
iRST	0	Signal	Internal
reg_WE	0	Signal	Internal
mem_WE	1	Signal	Internal
MemtoReg	1	Signal	Internal
ALUSrc	1	Signal	Internal
nAdd_Sub	0	Signal	Internal
ext0_or_S	1	Signal	Internal
rs	5'h1A	Signal	Internal
rt	5'h01	Signal	Internal
rd	5'h01	Signal	Internal
iImm	16'h00...	Signal	Internal
result	32'h00...	Signal	Internal
cCLK_PER	60 ns	Cons...	Internal

Wave - Default

	Msgs
iCLK	1
iRST	0
reg_WE	0
mem_WE	1
MemtoReg	1
ALUSrc	1
nAdd_Sub	0
ext0_or_S	1
rs	5'h1A
rt	5'h01
rd	5'h01
iImm	16'h0000
result	32'h00000003



Objects

Name	Value	Kind	
gCLK_HPER	30 ns	Gene...	In
DATA_WIDTH	32'h20	Gene...	In
ADDR_WIDTH	32'hA	Gene...	In
N	32'h20	Gene...	In
iCLK	0	Signal	Internal
iRST	0	Signal	Internal
reg_WE	1	Signal	Internal
mem_WE	0	Signal	Internal
MemtoReg	0	Signal	Internal
ALUSrc	0	Signal	Internal
nAdd_Sub	0	Signal	Internal
ext0_or_S	0	Signal	Internal
rs	5'h01	Signal	Internal
rt	5'h02	Signal	Internal
rd	5'h01	Signal	Internal
iImm	16'h00...	Signal	Internal
result	32'h00...	Signal	Internal
cCLK_PER	60 ns	Cons...	Internal

Wave - Default

	Msgs
iCLK	0
iRST	0
reg_WE	1
mem_WE	0
MemtoReg	0
ALUSrc	0
nAdd_Sub	0
ext0_or_S	0
rs	5'h01
rt	5'h02
rd	5'h01
iImm	16'h0000
result	32'h00000006

Part5c

This testbench is designed to verify the functionality of a Simplified MIPS Processor Datapath by simulating basic arithmetic, memory load/store, and register operations.

Reset and Initialization: Resets all registers and memory. Signals align to ensure proper setup.

Memory Initialization: Memory addresses 0, 1, and 2 store `0x00000010`, `0x00000020`, and `0x00000030`, respectively.

Immediate Addition (`addi`): `$25` gets `0`; `$26` gets `256`. Control signals enable immediate operations.

Load Word (`lw`): `$1` loads from address `0`, `$2` from address `4`. Memory data transfer aligns with control signals.

Register Addition (`add`): `$1` adds `$1 + $2` to get `0x00000030`. Registers operate as expected.

Store Word (`sw`): `$1` (`0x00000030`) is saved to memory at address `256`.

Load Word (`lw`): `$2` fetches from address `8`. Value matches memory data.

Register Addition (`add`): `$1` computes `$1 + $2`, producing `0x00000060`.