1. Design of Full Adder.

Full Adder Module Code

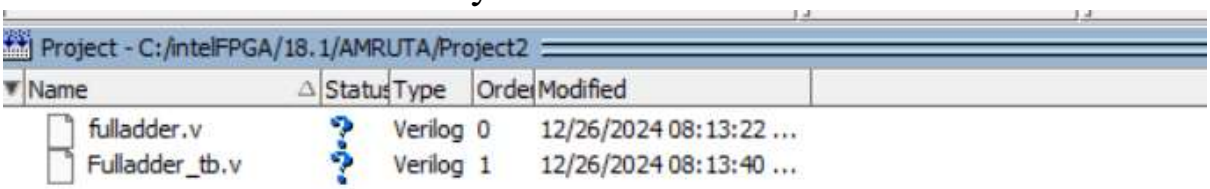
```
//full adder using Verilog Operator
module full_adder_o (
    input a,b,cin,
    output sum,carry
);

assign {carry,sum} = a+b+cin;

endmodule
```

Full Adder Test Bench Code

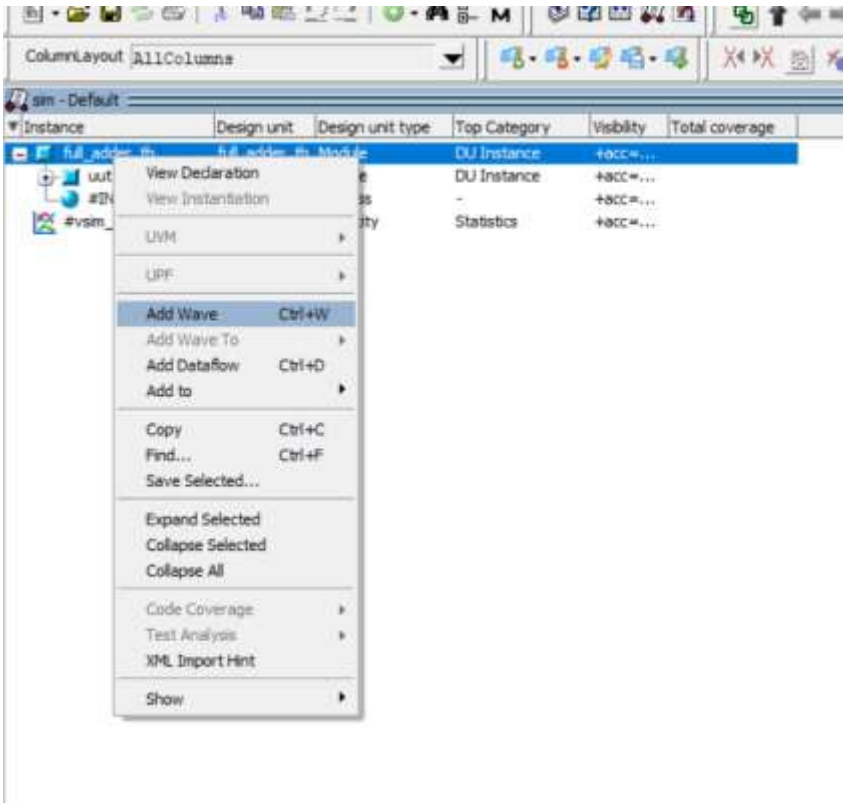
```
module full_adder_tb;
reg a,b,cin;
wire sum,carry;

full_adder_o uut (a,b,cin,sum,carry);

initial begin
a = 0; b = 0; cin = 0;
#10
a = 0; b = 0; cin = 1;
#10
a = 0; b = 1; cin = 0;
#10
a = 0; b = 1; cin = 1;
#10
a = 1; b = 0; cin = 0;
#10
a = 1; b = 0; cin = 1;
#10
a = 1; b = 1; cin = 0;
#10
a = 1; b = 1; cin = 1;
#10
$finish();
end

endmodule
```

Execution Process

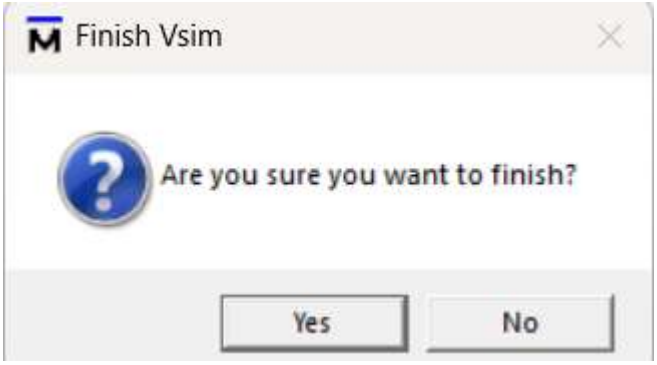
a. Firstly create a new project
b. Create both the files module file and test bench file.
c. Once both files are created you can see this



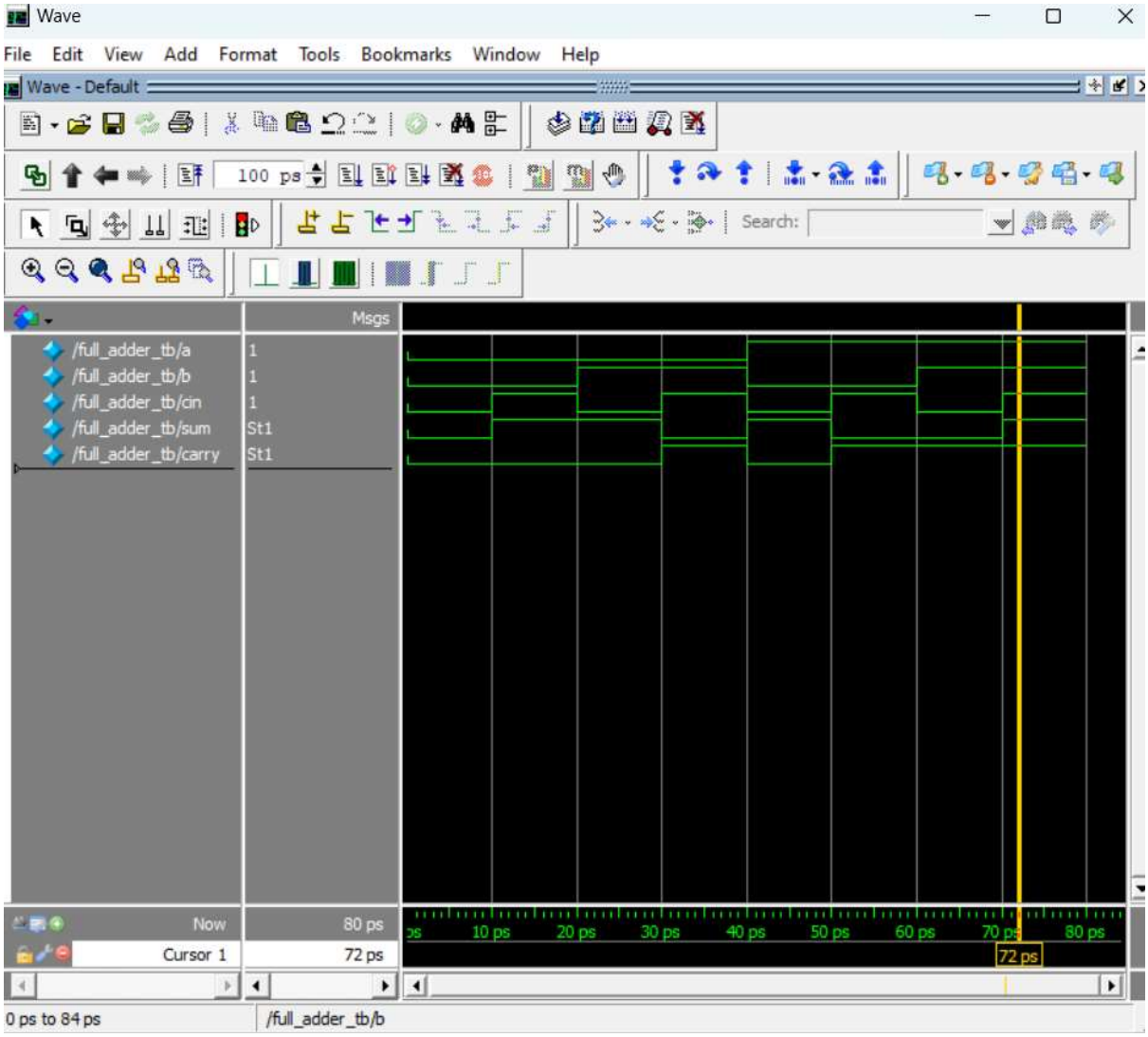
d. Double click on the file name (fulladder.v) a editor window will open on right side.
e. Type the module code and save it.
f. Similarly process with 2nd file fulladder_tb.v, type the testbench file and save it.
g. Then right click on question mark of each fil and select compile selected. If the file compiles the file with no errors.
h. Same for 2nd file.
i. Click on Library, select work folder (click on + icon)
j. Select the file(fulladder_tb.v), right click and simulate.
k. Then from the below sim window select add wave



l. A wave window will appear, back on the simulator windows click on run



m. Click on No, and check back the wave form window for Final Output.



2. Design of the half adder.

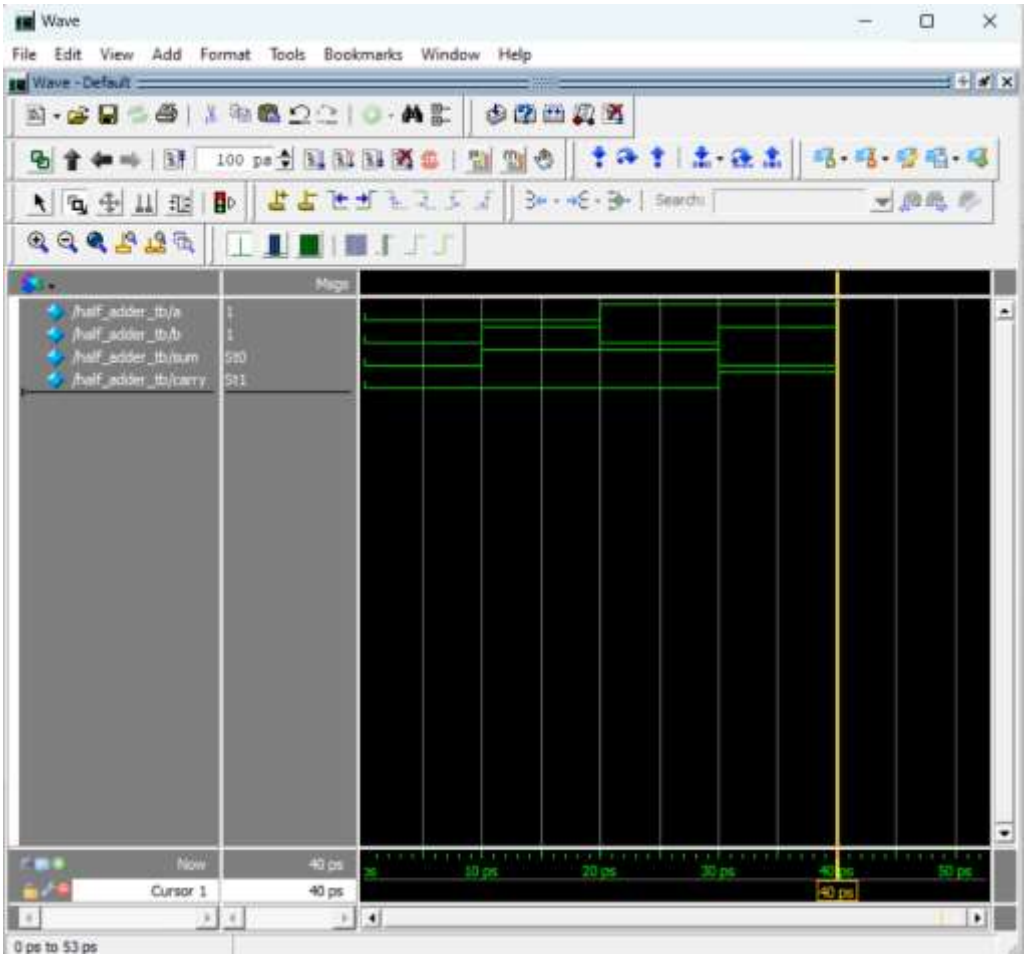
Half adder module

```
Ln#
1  module half_adder_s (
2      input a,b,
3      output sum,carry
4  );
5
6  xor(sum,a,b);
7  and(carry,a,b);
8
9  endmodule
10
```

Half adder Test Bench

```
Ln#
1  module half_adder_tb;
2  reg a,b;
3  wire sum,carry;
4
5  half_adder_s uut(a,b,sum,carry);
6
7  initial begin
8  a = 0; b = 0;
9  #10
10 b = 0; b = 1;
11 #10
12 a = 1; b = 0;
13 #10
14 b = 1; b = 1;
15 #10
16 $finish();
17 end
18
19 endmodule
20
```

OUTPUT



3. Realization of a Boolean Function. Minimize using K map and realize the same using truth table

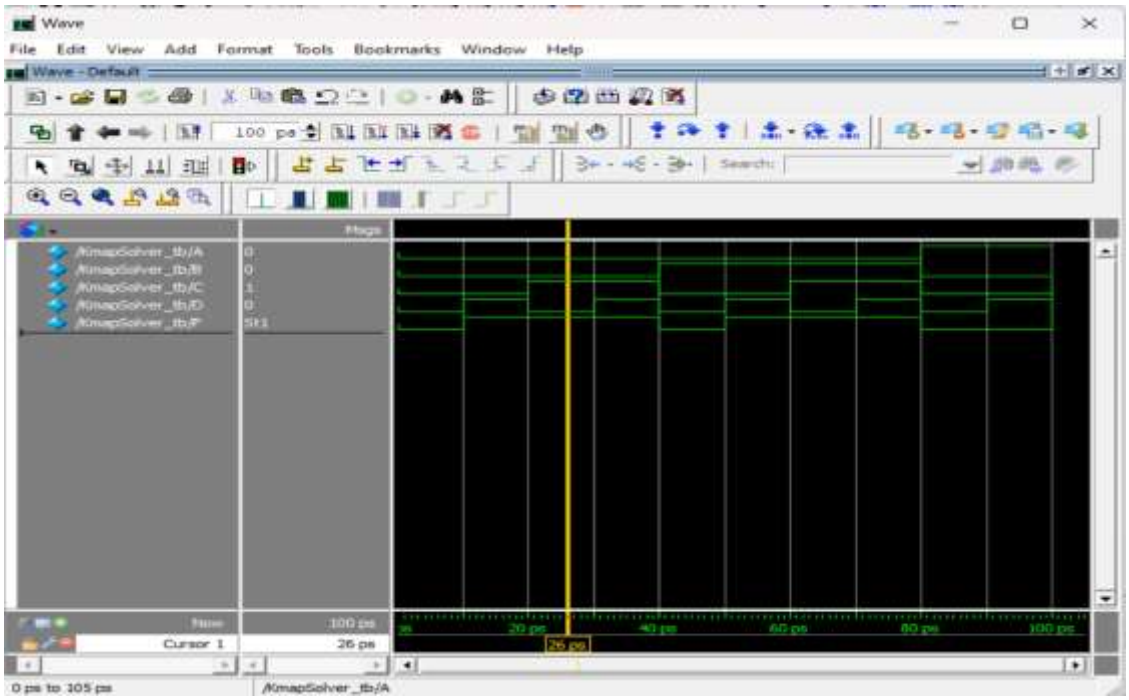
C:/intelFPGA/18.1/AMRUTA/kmap.v (/KmapSolver_tb/uut) - Default

```
module KmapSolver(
    input A, B, C, D,
    output F
);
    assign F = (~A & ~C & D) | (C & D) | (~B & C) | (B & C & ~D); // Correct Boolean logic
endmodule
```

C:/intelFPGA/18.1/AMRUTA/kmaptb.v - Default

```
module KmapSolver_tb;

    // Inputs
    reg A, B, C, D;

    // Output
    wire F;

    // Instantiate the Unit Under Test (UUT)
    KmapSolver uut (
        .A(A),
        .B(B),
        .C(C),
        .D(D),
        .F(F)
    );

    initial begin
        $monitor("Time = %0t | A = %b, B = %b, C = %b, D = %b | F = %b",
                 $time, A, B, C, D, F);

        // Apply all 16 combinations of A, B, C, and D
        A = 0; B = 0; C = 0; D = 0; #10;
        A = 0; B = 0; C = 0; D = 1; #10;
        A = 0; B = 0; C = 1; D = 0; #10;
        A = 0; B = 0; C = 1; D = 1; #10;
        A = 0; B = 1; C = 0; D = 0; #10;
        A = 0; B = 1; C = 0; D = 1; #10;
        A = 0; B = 1; C = 1; D = 0; #10;
        A = 0; B = 1; C = 1; D = 1; #10;
        A = 1; B = 0; C = 0; D = 0; #10;
        A = 1; B = 0; C = 0; D = 1; #10;
        A = 1; B = 0; C = 1; D = 0; #10;
        A = 1; B = 0; C = 1; D = 1; #10;
        A = 1; B = 1; C = 0; D = 0; #10;
        A = 1; B = 1; C = 0; D = 1; #10;
        A = 1; B = 1; C = 1; D = 0; #10;
        A = 1; B = 1; C = 1; D = 1; #10;

        $finish;
    end

endmodule
```

OUTPUT



Truth Table for given example is F(A,B,C,D)=Σm(0,2,5,7,8,10,13,15)

A	B	C	D	F
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

4. Realize NAND and NOR Gate as universal gate

1. NAND Implementation:
   - NOT Gate: $\mathrm{NOT}(A) = \mathrm{NAND}(A, A)$
   - AND Gate: $\mathrm{AND}(A, B) = \mathrm{NOT}(\mathrm{NAND}(A, B))$
   - OR Gate: $\mathrm{OR}(A, B) = \mathrm{NAND}(\mathrm{NOT}(A), \mathrm{NOT}(B))$
2. NOR Implementation:
   - NOT Gate: $\mathrm{NOT}(A) = \mathrm{NOR}(A, A)$
   - OR Gate: $\mathrm{OR}(A, B) = \mathrm{NOT}(\mathrm{NOR}(A, B))$
   - AND Gate: $\mathrm{AND}(A, B) = \mathrm{NOT}(\mathrm{NOR}(\mathrm{NOT}(A), \mathrm{NOT}(B)))$

MAIN FUNCTION CODE

C:/intelFPGA/18.1/AMRUTA/nand.v - Default

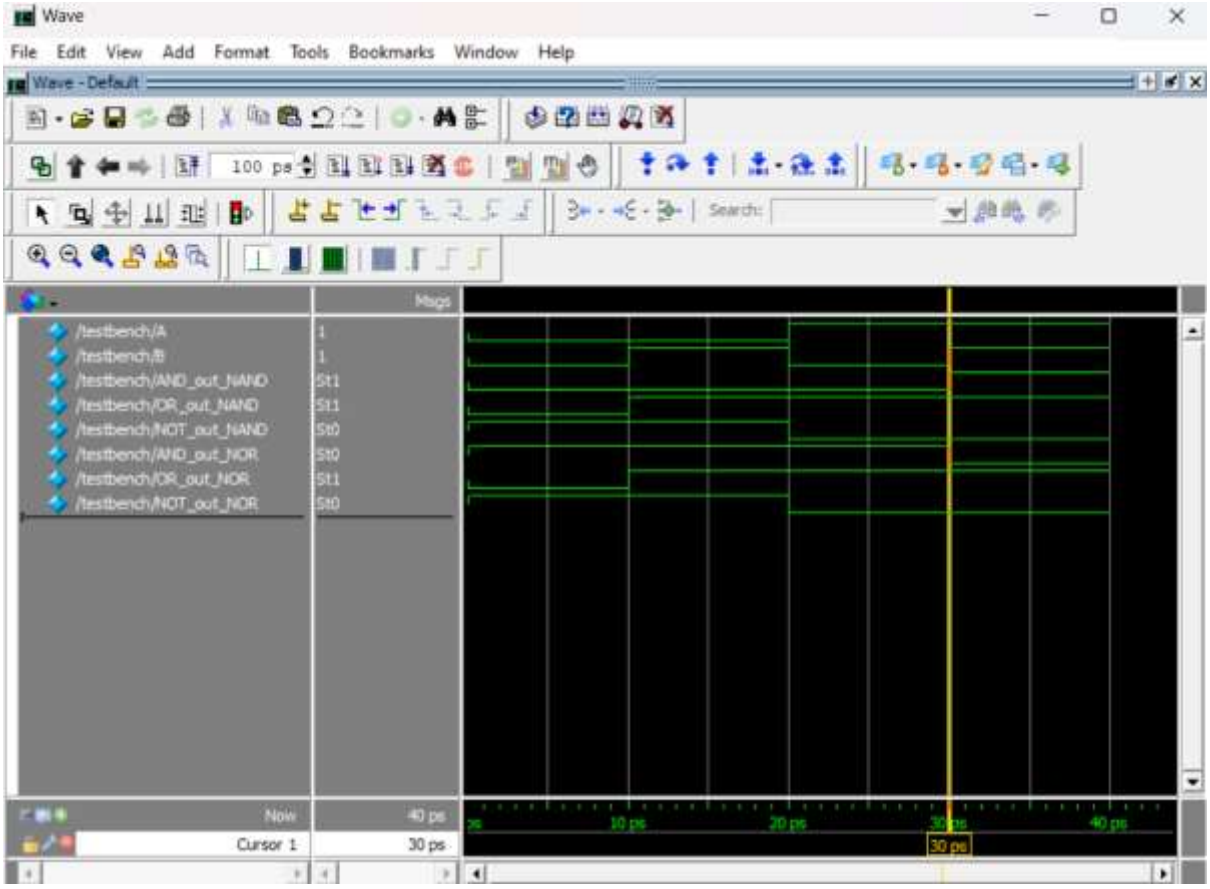
```
// NAND Gate as Universal Gate
module NAND_as_Universal(
    input A, B,
    output AND_out, OR_out, NOT_out
);
    wire NAND1, NAND2, NAND3;

    // NOT gate using NAND
    assign NOT_out = ~(A & A);

    // AND gate using NAND
    assign NAND1 = ~(A & B);
    assign AND_out = ~(NAND1 & NAND1);

    // OR gate using NAND
    assign NAND2 = ~(A & A);
    assign NAND3 = ~(B & B);
    assign OR_out = ~(NAND2 & NAND3);
endmodule

// NOR Gate as Universal Gate
module NOR_as_Universal(
    input A, B,
    output AND_out, OR_out, NOT_out
);
    wire NOR1, NOR2, NOR3;

    // NOT gate using NOR
    assign NOT_out = ~(A | A);

    // OR gate using NOR
    assign OR_out = ~(~(A | B));

    // AND gate using NOR
    assign NOR1 = ~(A | A);
    assign NOR2 = ~(B | B);
    assign NOR3 = ~(NOR1 | NOR2);
    assign AND_out = ~(NOR3 | NOR3);
endmodule
```

TESTBENCH

C:/intelFPGA/18.1/AMRUTA/nandnortb.v (/testbench) - Default

```
// Testbench to verify NAND and NOR as Universal Gates
module testbench;
    reg A, B;
    wire AND_out_NAND, OR_out_NAND, NOT_out_NAND;
    wire AND_out_NOR, OR_out_NOR, NOT_out_NOR;

    // Instantiate NAND as Universal
    NAND_as_Universal NAND_U(
        .A(A), .B(B),
        .AND_out(AND_out_NAND),
        .OR_out(OR_out_NAND),
        .NOT_out(NOT_out_NAND)
    );

    // Instantiate NOR as Universal
    NOR_as_Universal NOR_U(
        .A(A), .B(B),
        .AND_out(AND_out_NOR),
        .OR_out(OR_out_NOR),
        .NOT_out(NOT_out_NOR)
    );

    initial begin
        $monitor("A=%b B=%b | NAND: AND=%b OR=%b NOT=%b | NOR: AND=%b OR=%b NOT=%b",
                 A, B, AND_out_NAND, OR_out_NAND, NOT_out_NAND,
                 AND_out_NOR, OR_out_NOR, NOT_out_NOR);

        // Test inputs
        A = 0; B = 0; #10;
        A = 0; B = 1; #10;
        A = 1; B = 0; #10;
        A = 1; B = 1; #10;

        $finish;
    end
endmodule
```

OUTPUT



5. DESIGN A FULL ADDER USING TWO HALF ADDER CIRCUITS.

MAIN FUNCTION

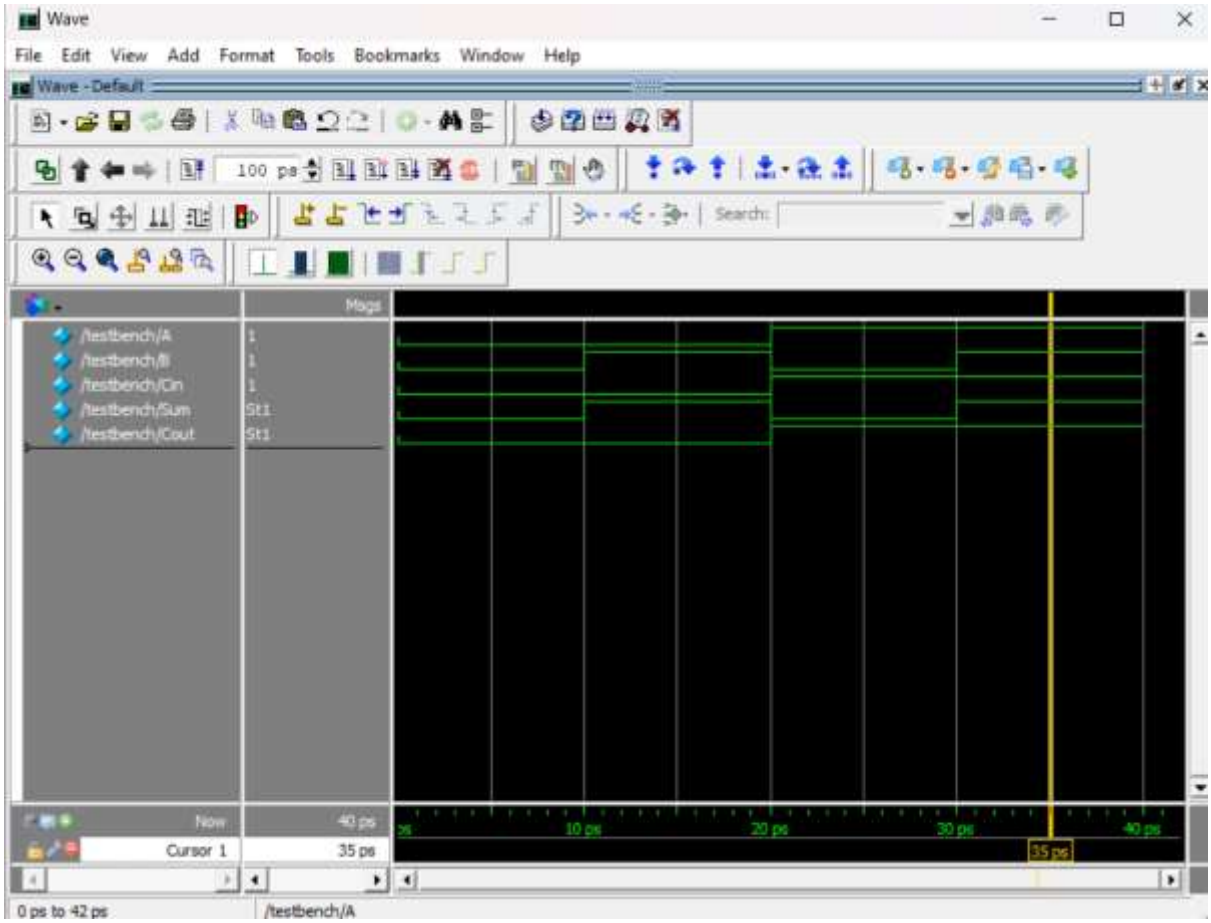
```
// Half Adder Module
module HalfAdder(
    input A, B,
    output Sum, Carry
);
    assign Sum = A ^ B;  // XOR for Sum
    assign Carry = A & B; // AND for Carry
endmodule

// Full Adder Module Using Two Half Adders
module FullAdder(
    input A, B, Cin,
    output Sum, Cout
);
    wire S1, C1, C2;

    // First Half Adder
    HalfAdder HA1 (
        .A(A), .B(B),
        .Sum(S1), .Carry(C1)
    );

    // Second Half Adder
    HalfAdder HA2 (
        .A(S1), .B(Cin),
        .Sum(Sum), .Carry(C2)
    );

    // Final Carry Out
    assign Cout = C1 | C2;
endmodule
```

TEST BENCH

C:/intelFPGA/18.1/AMRUTA/FAU2HATB.V (/testbench) - Default

```
module testbench;
    reg A, B, Cin;
    wire Sum, Cout;

    // Instantiate Full Adder
    FullAdder FA (
        .A(A), .B(B), .Cin(Cin),
        .Sum(Sum), .Cout(Cout)
    );

    initial begin
        $monitor("A=%b B=%b Cin=%b | Sum=%b Cout=%b", A, B, Cin, Sum, Cout);

        // Test cases
        A = 0; B = 0; Cin = 0; #10;
        A = 0; B = 1; Cin = 0; #10;
        A = 1; B = 0; Cin = 1; #10;
        A = 1; B = 1; Cin = 1; #10;

        $finish;
    end
endmodule
```

OUTPUT



6. DESIGN A FULL SUBSTRACTOR USING TWO HALF SUBTRACTOR

Explanation

1. **Half Subtractor 1:**
   - Computes $\text{Diff1} = A \oplus B$ (intermediate difference).
   - Computes $\text{Borrow1} = \sim A \cdot B$ (borrow from $A - B$).
2. **Half Subtractor 2:**
   - Computes $\text{Diff} = \text{Diff1} \oplus \text{Bin}$ (final difference).
   - Computes $\text{Borrow2} = \sim \text{Diff1} \cdot \text{Bin}$ (borrow from $\text{Diff1} - \text{Bin}$).
3. **Final Borrow (Bout):**
   - Combines the borrows from both Half Subtractors: $\text{Bout} = \text{Borrow1} | \text{Borrow2}$.

MAIN FUNCTION

C:/intelFPGA/18.1/AMRUTA/FSU2HS.V (/testbench/FS)

File Edit View Tools Bookmarks Window Help

C:/intelFPGA/18.1/AMRUTA/FSU2HS.V (/testbench/FS) - Default

Ln#

```
// Half Subtractor Module
module HalfSubtractor(
    input A, B,
    output Diff, Borrow
);
    assign Diff = A ^ B;       // XOR for Difference
    assign Borrow = ~A & B;    // Borrow when A < B
endmodule

// Full Subtractor Module Using Two Half Subtractors
module FullSubtractor(
    input A, B, Bin,     // Inputs: A, B, Borrow-in (Bin)
    output Diff, Bout    // Outputs: Difference, Borrow-out (Bout)
);
    wire D1, B1, B2;

    // First Half Subtractor
    HalfSubtractor HS1 (
        .A(A), .B(B),
        .Diff(D1), .Borrow(B1)
    );

    // Second Half Subtractor
    HalfSubtractor HS2 (
        .A(D1), .B(Bin),
        .Diff(Diff), .Borrow(B2)
    );

    // Final Borrow Out
    assign Bout = B1 | B2;     // Bout is the OR of both borrows
endmodule
```

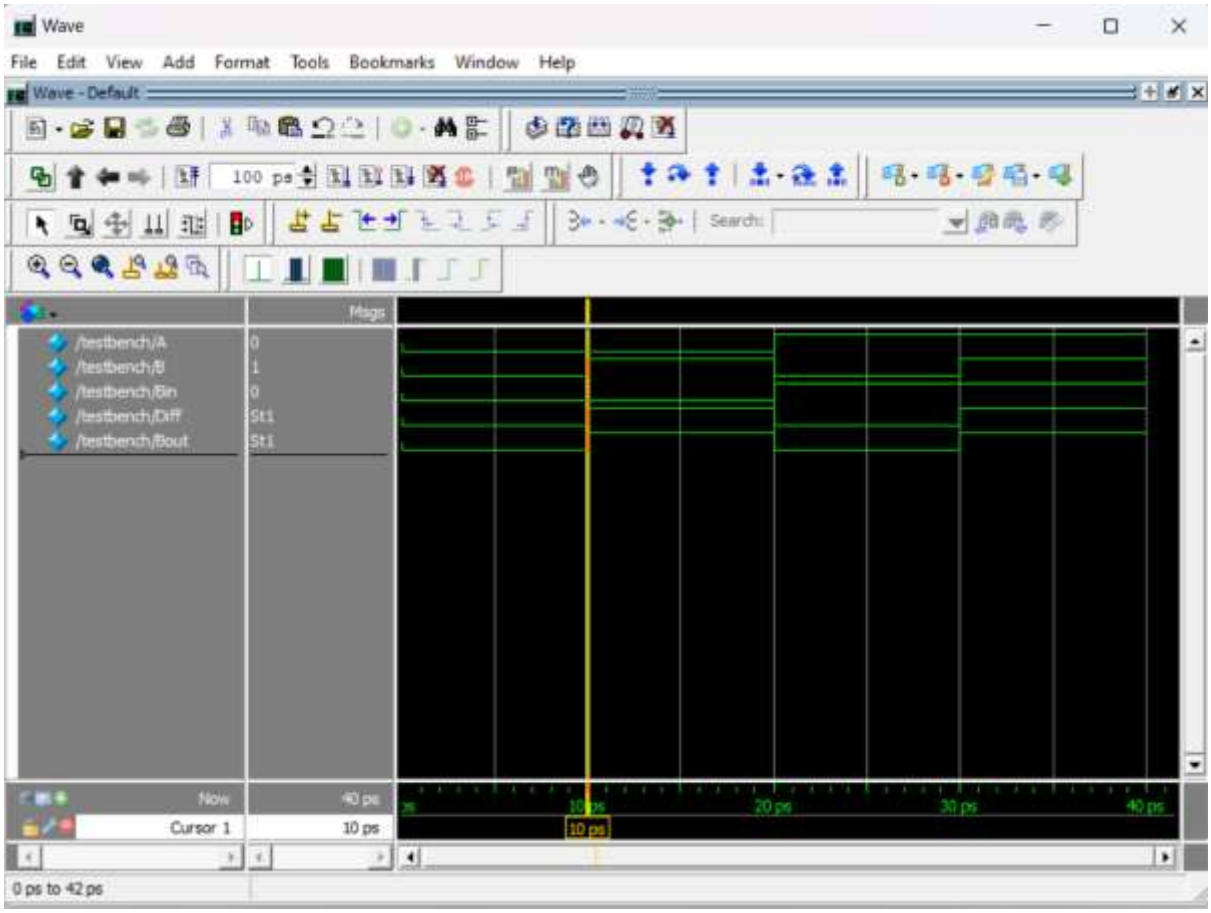
TEST BENCH CODE

C:/intelFPGA/18.1/AMRUTA/FSU2HSTB.V (/testbench)

File Edit View Tools Bookmarks Window Help

```
module testbench;
    reg A, B, Bin;
    wire Diff, Bout;

    // Instantiate Full Subtractor
    FullSubtractor FS (
        .A(A), .B(B), .Bin(Bin),
        .Diff(Diff), .Bout(Bout)
    );

    initial begin
        $monitor("A=%b B=%b Bin=%b | Diff=%b Bout=%b", A, B, Bin, Diff, Bout);

        // Test cases
        A = 0; B = 0; Bin = 0; #10;
        A = 0; B = 1; Bin = 0; #10;
        A = 1; B = 0; Bin = 1; #10;
        A = 1; B = 1; Bin = 1; #10;

        $finish;
    end
endmodule
```

OUTPUT



7. Design Full Subtractor WITHOUT using Half Subtractor

1. **Difference** (Diff):
   - Computed using $A \oplus B \oplus \text{Bin}$, which gives the correct difference for all input cases.
2. **Borrow-Out** (Bout):
   - Borrow occurs in two scenarios:
     1. When B is greater than A: $\sim A \cdot B$
     2. When Bin causes an additional borrow: $(\sim (A \oplus B)) \cdot \text{Bin}$
   - Combine these cases using OR: $\text{Bout} = (\sim A \cdot B) | ((\sim (A \oplus B)) \cdot \text{Bin})$.
3. **Testbench:**
   - Tests all combinations of A, B, and Bin to validate the design.

MAIN CODE FOR FULL SUBSTRACTOR

C:/intelFPGA/18.1/AMRUTA/FS.V (/testbench/FS)

File Edit View Tools Bookmarks Window Help

C:/intelFPGA/18.1/AMRUTA/FS.V (/testbench/FS) - Default

Ln#

```
// Full Subtractor Module
module FullSubtractor(
    input A, B, Bin,        // Inputs: A, B, Borrow-in (Bin)
    output Diff, Bout       // Outputs: Difference and Borrow-out (Bout)
);
    // Directly compute the difference and borrow-out
    assign Diff = A ^ B ^ Bin;              // XOR for Difference
    assign Bout = (~A & B) | ((~(A ^ B)) & Bin); // Borrow-out logic
endmodule
```

TESTBENCH CODE

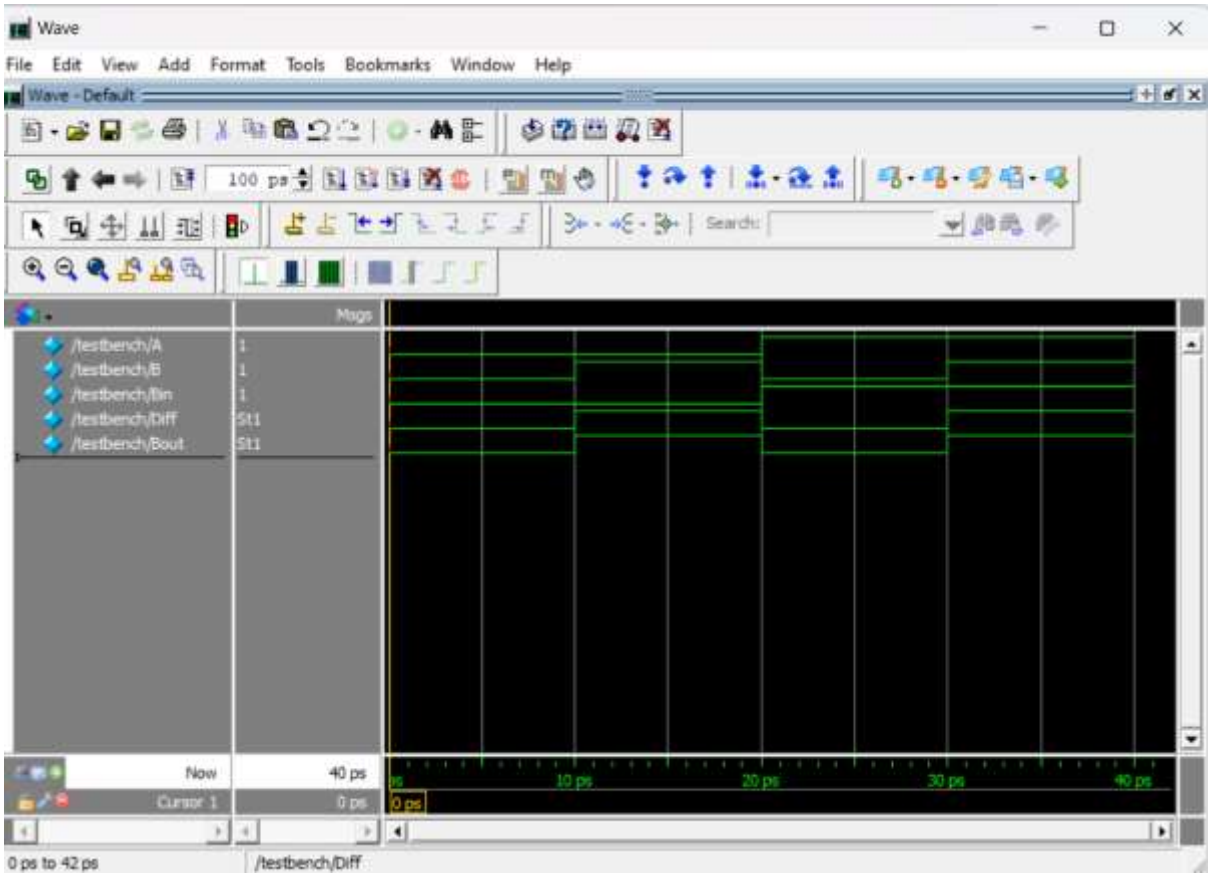
C:/intelFPGA/18.1/AMRUTA/FSTB.V (/testbench)

File Edit View Tools Bookmarks Window Help

C:/intelFPGA/18.1/AMRUTA/FSTB.V (/testbench) - Default

```
module testbench;
    reg A, B, Bin;
    wire Diff, Bout;

    // Instantiate Full Subtractor
    FullSubtractor FS (
        .A(A), .B(B), .Bin(Bin),
        .Diff(Diff), .Bout(Bout)
    );

    initial begin
        $monitor("A=%b B=%b Bin=%b | Diff=%b Bout=%b", A, B, Bin, Diff, Bout);

        // Test cases
        A = 0; B = 0; Bin = 0; #10;
        A = 0; B = 1; Bin = 0; #10;
        A = 1; B = 0; Bin = 1; #10;
        A = 1; B = 1; Bin = 1; #10;

        $finish;
    end
endmodule
```

OUTPUT



8. Design 4 bit parallel Adder Subtractor Composite unit using IC7483 and 7486

Explanation

1. **Addition Mode** (`Mode = 0`):
   - The input B passes through unmodified to the adder.
   - $\text{CarryIn} = 0$.
2. **Subtraction Mode** (`Mode = 1`):
   - The input B is XORed with Mode, flipping its bits to create B'.
   - $\text{CarryIn} = 1$ adds the additional 1 for 2's complement subtraction.
3. **Output Calculation:**
   - For addition: $\text{Result} = A + B$.
   - For subtraction: $\text{Result} = A + B' + 1$.
4. **Simulation in ModelSim:**
   - Run the testbench in ModelSim. It applies multiple test cases and verifies addition and subtraction behavior.

IC 7483 and 7486 Mapping

- **7483 (4-bit binary adder):**
  - Simulated using the addition logic in Verilog.
- **7486 (XOR gate):**
  - XOR gates are implemented in the code for generating B' when performing subtraction.

Main Module Code

C:/intelFPGA/18.1/AMRUTA/4bitparadd.v - Default

```
module AdderSubtractor4Bit(
    input [3:0] A, B,        // 4-bit inputs A and B
    input Mode,              // Mode: 0 for Addition, 1 for Subtraction
    output [3:0] Result,     // 4-bit result
    output Carry             // Carry out
);
    wire [3:0] B_XOR;        // XORed version of B for subtraction
    wire CarryIn;            // Carry-In for addition/subtraction

    // XOR B with Mode to perform 2's complement in subtraction mode
    assign B_XOR = B ^ {4{Mode}};
    assign CarryIn = Mode;  // Carry-In is 1 for subtraction (to add 2's complement)

    // 4-bit binary adder (IC 7483 implementation)
    assign {Carry, Result} = A + B_XOR + CarryIn;
endmodule
```

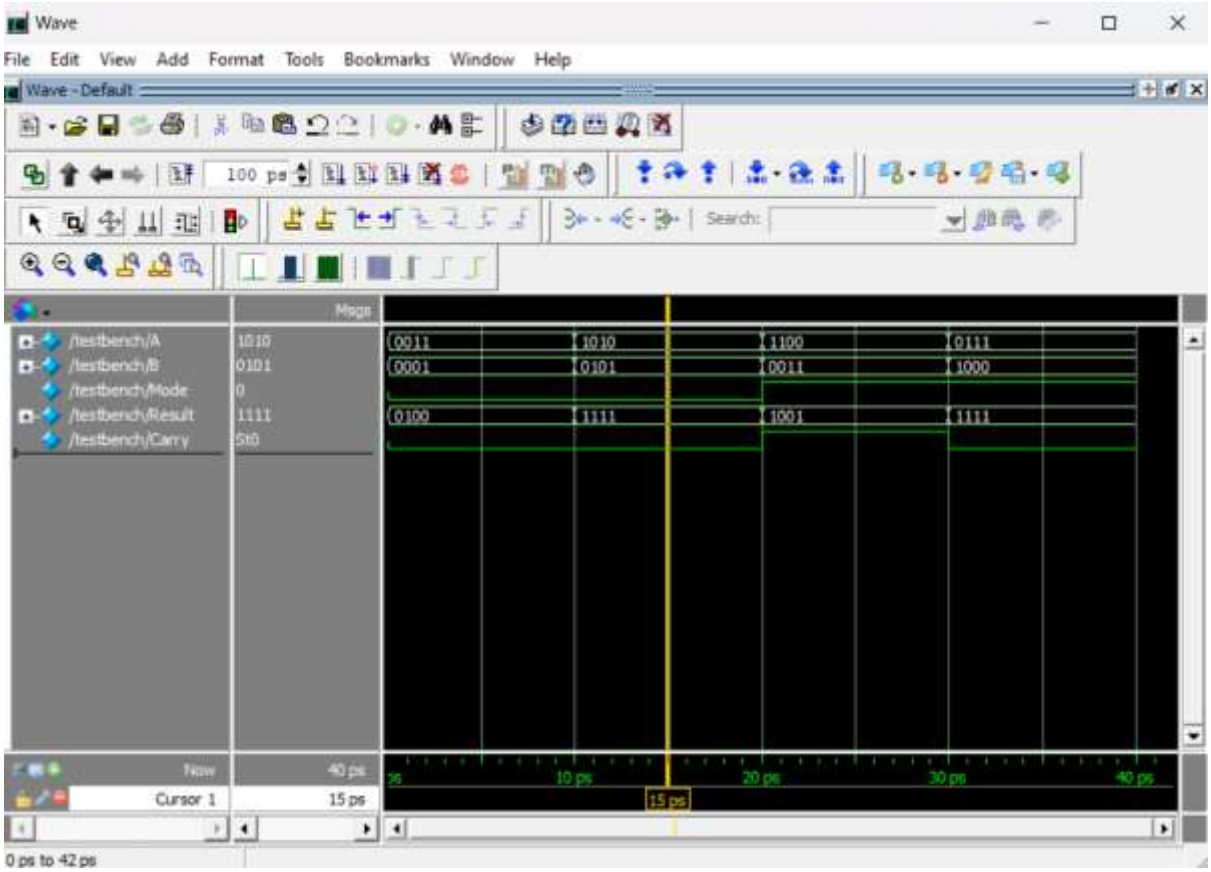
Testbench Code

File Edit View Tools Bookmarks Window Help

C:/intelFPGA/18.1/AMRUTA/4bitparaddtb.v - Default

```
module testbench;
    reg [3:0] A, B;          // 4-bit inputs A and B
    reg Mode;                // Mode: 0 for Addition, 1 for Subtraction
    wire [3:0] Result;       // 4-bit result
    wire Carry;              // Carry out

    // Instantiate the Adder-Subtractor
    AdderSubtractor4Bit UUT (
        .A(A),
        .B(B),
        .Mode(Mode),
        .Result(Result),
        .Carry(Carry)
    );

    initial begin
        $monitor("Time=%0t | A=%b B=%b Mode=%b | Result=%b Carry=%b",
                 $time, A, B, Mode, Result, Carry);

        // Test cases
        A = 4'b0011; B = 4'b0001; Mode = 0; #10; // Add: 3 + 1
        A = 4'b1010; B = 4'b0101; Mode = 0; #10; // Add: 10 + 5
        A = 4'b1100; B = 4'b0011; Mode = 1; #10; // Sub: 12 - 3
        A = 4'b0111; B = 4'b1000; Mode = 1; #10; // Sub: 7 - 8

        $finish;
    end
endmodule
```

OUTPUT



9. Design 8:1 Multiplexer using two 4:1 Multiplexer.

Explanation

1. **4:1 MUX Modules:**
   - Each **4:1 MUX** selects one of its 4 inputs based on two select lines (`Sel[1:0]`).
2. **2:1 MUX Module:**
   - Combines the outputs of the two **4:1 MUXes** based on the most significant select line (`Sel[2]`).
3. **8:1 MUX:**
   - The **8:1 MUX** uses two **4:1 MUXes** to process the first and second sets of 4 inputs.
   - A **2:1 MUX** selects between the outputs of the two **4:1 MUXes**.
4. **Testbench:**
   - Tests all 8 select lines to verify that the correct data is output.

MAIN FUNCTION CODE

C:/intelFPGA/18.1/AMRUTA/81MULTIPLEXER.V - Default

```
// 4:1 Multiplexer Module
module Mux4to1(
    input [3:0] D,   // Data inputs
    input [1:0] Sel, // Select lines
    output Y         // Output
);
    assign Y = (Sel == 2'b00) ? D[0] :
               (Sel == 2'b01) ? D[1] :
               (Sel == 2'b10) ? D[2] :
                                D[3];
endmodule

// 2:1 Multiplexer Module
module Mux2to1(
    input A, B, Sel,
    output Y
);
    assign Y = (Sel == 1'b0) ? A : B;
endmodule

// 8:1 Multiplexer Module Using Two 4:1 Multiplexers and a 2:1 Multiplexer
module Mux8to1(
    input [7:0] D,    // 8 Data inputs
    input [2:0] Sel,  // 3 Select lines
    output Y          // Output
);
    wire Y0, Y1;      // Outputs of the two 4:1 Muxes

    // Instantiate two 4:1 Muxes
    Mux4to1 Mux1 (
        .D(D[3:0]), .Sel(Sel[1:0]), .Y(Y0)
    );

    Mux4to1 Mux2 (
        .D(D[7:4]), .Sel(Sel[1:0]), .Y(Y1)
    );

    // Instantiate a 2:1 Mux to combine the results
    Mux2to1 Mux3 (
        .A(Y0), .B(Y1), .Sel(Sel[2]), .Y(Y)
    );
endmodule
```

TESTBENCH CODE

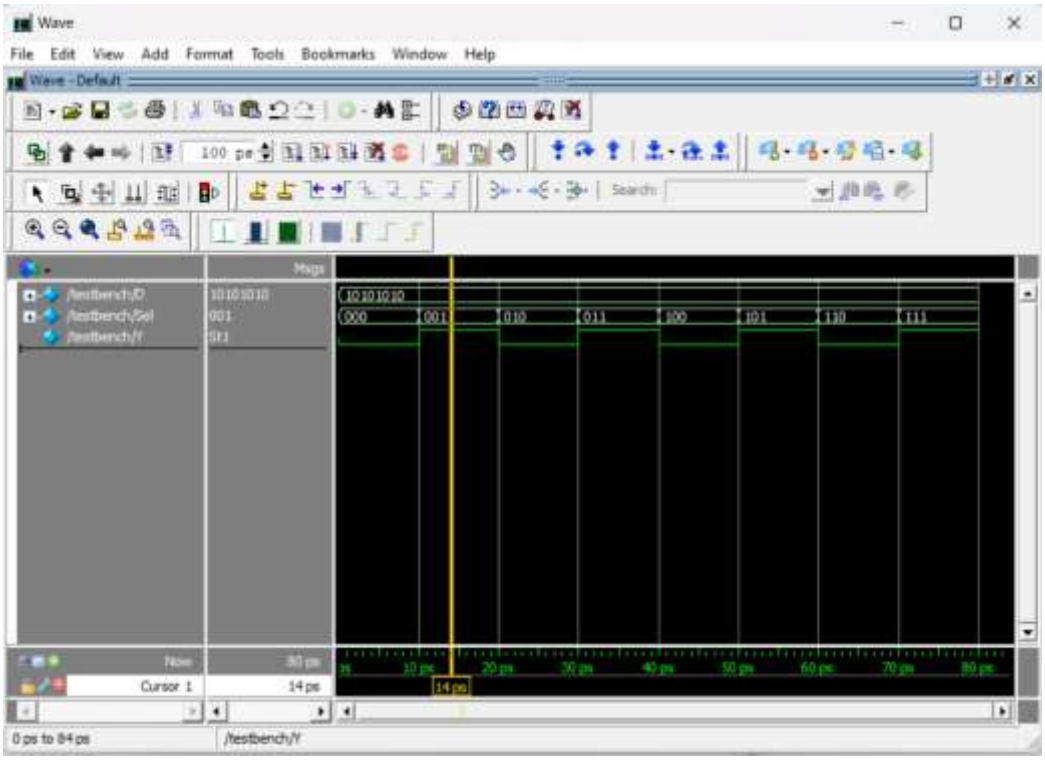
C:/intelFPGA/18.1/AMRUTA/81MULTIPLEXERTB.V (/testbench)

File Edit View Tools Bookmarks Window Help

C:/intelFPGA/18.1/AMRUTA/81MULTIPLEXERTB.V (/testbench) - Default

```
module testbench;
    reg [7:0] D;          // 8 Data inputs
    reg [2:0] Sel;        // 3 Select lines
    wire Y;               // Output

    // Instantiate 8:1 Multiplexer
    Mux8to1 UUT (
        .D(D),
        .Sel(Sel),
        .Y(Y)
    );

    initial begin
        $monitor("Time=%0t | D=%b Sel=%b | Y=%b", $time, D, Sel, Y);

        // Test cases
        D = 8'b10101010; Sel = 3'b000; #10; // Select D[0]
        D = 8'b10101010; Sel = 3'b001; #10; // Select D[1]
        D = 8'b10101010; Sel = 3'b010; #10; // Select D[2]
        D = 8'b10101010; Sel = 3'b011; #10; // Select D[3]
        D = 8'b10101010; Sel = 3'b100; #10; // Select D[4]
        D = 8'b10101010; Sel = 3'b101; #10; // Select D[5]
        D = 8'b10101010; Sel = 3'b110; #10; // Select D[6]
        D = 8'b10101010; Sel = 3'b111; #10; // Select D[7]

        $finish;
    end
endmodule
```

OUTPUT



10. Implement logic function using Multiplexer.

Concept of Implementing Logic Functions Using a Multiplexer

Multiplexers (MUX) can implement any logic function by using their **select lines** to determine which data input is passed to the output. The MUX acts like a "hardware look-up table" where combinations of inputs are pre-programmed to output the desired function.

Steps to Implement a Logic Function Using a MUX:

1. **Determine the number of variables in the logic function:**
   - A logic function with n variables requires a 2^n-to-1 multiplexer. Alternatively, smaller multiplexers (like 4:1) can be used with some external logic.
2. **Assign variables to the select lines:**
   - The n inputs to the logic function are mapped to the MUX's select lines.
3. **Determine the MUX data inputs (truth table):**
   - Evaluate the logic function for each combination of the select lines.
   - These outputs become the MUX data inputs (D).
4. **Implement the logic using the MUX:**
   - The MUX selects the correct output for each input combination, thereby reproducing the logic function.

Example: Implement $F(A, B, C) = AB + AC$ Using a 4:1 Multiplexer

Step 1: Express the Function in Canonical Form

The function $F(A, B, C) = \overline{A}B + AC$ can be rewritten as:

$$F = \sum(2, 3, 6, 7)$$

This means F is 1 for $ABC = 010, 011, 110, 111$.

Step 2: Use A, B as Select Lines and Simplify C:

- Select lines: A and B ($S1$ and $S0$).
- Input C determines the MUX data inputs D.

Evaluate $F(A, B, C)$ for all A, B combinations:

A	B	C	F (Truth Table)
0	0	×	0
0	1	×	1
1	0	×	C
1	1	×	1

Step 3: Assign D Inputs for the MUX

From the truth table:

- $D[0] = 0$
- $D[1] = 1$
- $D[2] = C$
- $D[3] = 1$

MAIN FUNCTION CODE

C:/intelFPGA/18.1/AMRUTA/41mulimp.v

File Edit View Tools Bookmarks Window Help

C:/intelFPGA/18.1/AMRUTA/41mulimp.v - Default

```
module LogicFunctionUsingMUX(
    input A, B, C,
    output F
);
    wire [3:0] D;

    // Assign MUX inputs based on simplified function
    assign D[0] = 0;    // When A=0, B=0
    assign D[1] = 1;    // When A=0, B=1
    assign D[2] = C;    // When A=1, B=0
    assign D[3] = 1;    // When A=1, B=1

    // MUX implementation
    assign F = (A == 0 && B == 0) ? D[0] :
               (A == 0 && B == 1) ? D[1] :
               (A == 1 && B == 0) ? D[2] : D[3];
endmodule
```

TESTBENCH CODE

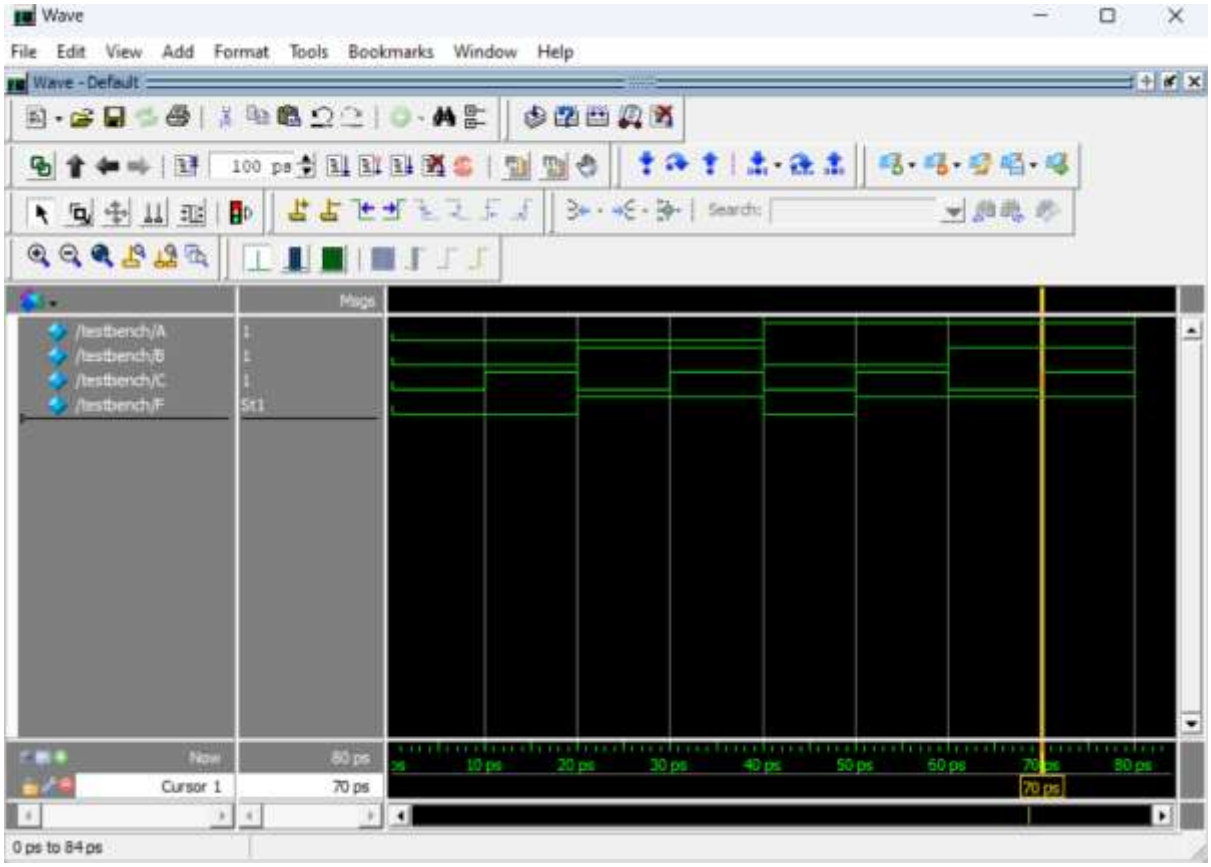
C:/intelFPGA/18.1/AMRUTA/41muximptb.v (/testbench)

File Edit View Tools Bookmarks Window Help

C:/intelFPGA/18.1/AMRUTA/41muximptb.v (/testbench) - Default

```
module testbench;
    reg A, B, C;      // Inputs to the logic function
    wire F;           // Output of the logic function

    // Instantiate the LogicFunctionUsingMUX module
    LogicFunctionUsingMUX UUT (
        .A(A),
        .B(B),
        .C(C),
        .F(F)
    );

    initial begin
        $monitor("Time=%0t | A=%b B=%b C=%b | F=%b", $time, A, B, C, F);

        // Test all combinations of A, B, C
        A = 0; B = 0; C = 0; #10;
        A = 0; B = 0; C = 1; #10;
        A = 0; B = 1; C = 0; #10;
        A = 0; B = 1; C = 1; #10;
        A = 1; B = 0; C = 0; #10;
        A = 1; B = 0; C = 1; #10;
        A = 1; B = 1; C = 0; #10;
        A = 1; B = 1; C = 1; #10;

        $finish;
    end
endmodule
```

OUTPUT



11. 8-bit Addition, Multiplication, Division

Explanation

1. **Addition (A + B)**: A simple addition of two 8-bit numbers.
2. **Multiplication (A * B)**: Produces a 16-bit result to accommodate overflow.
3. **Division (A / B)**: Handles the possibility of division by zero with an error flag.
4. **Testbench**: Tests all operations with sample inputs and outputs results to the simulation log.

MAIN FUNCTION CODE

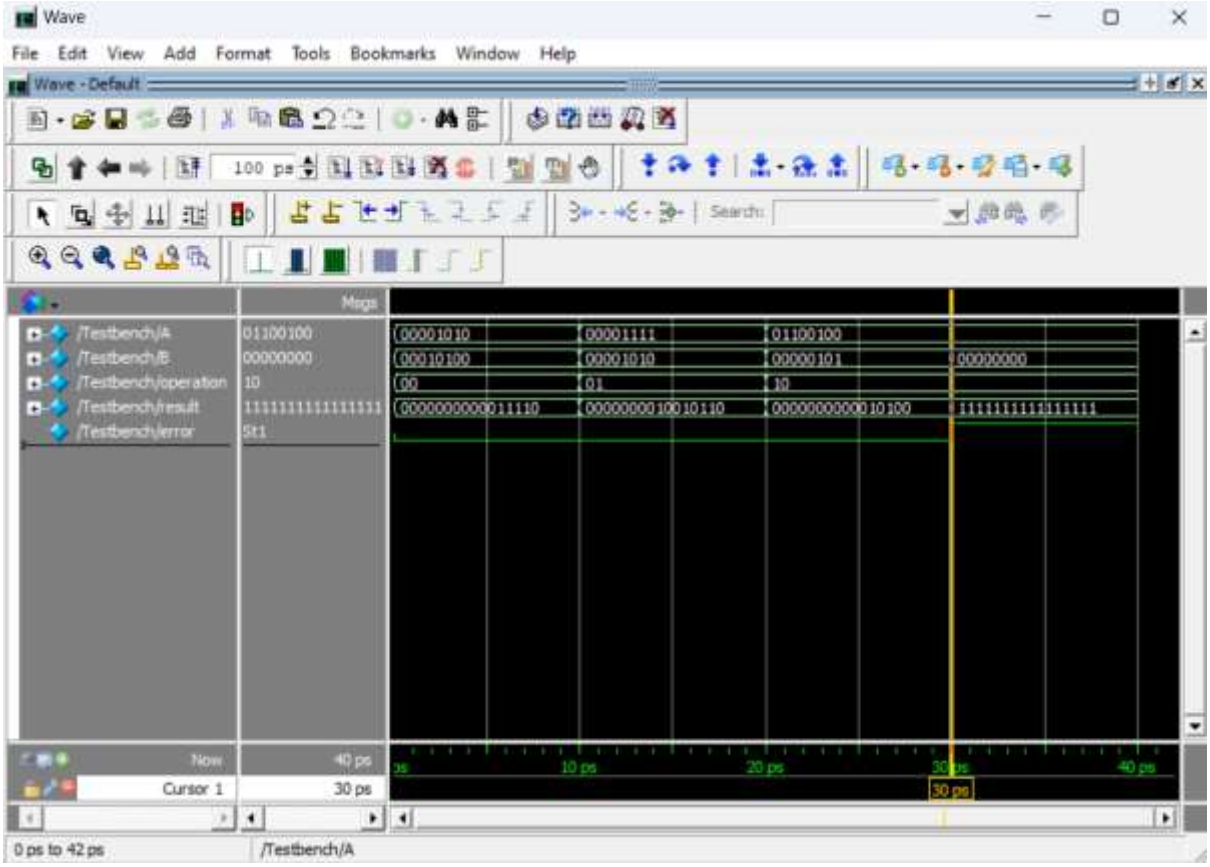
C:/intelFPGA/18.1/AMRUTA/8BITAMD.V

File Edit View Tools Bookmarks Window Help

C:/intelFPGA/18.1/AMRUTA/8BITAMD.V - Default

```
module ArithmeticOperations(
    input [7:0] A,          // 8-bit input A
    input [7:0] B,          // 8-bit input B
    input [1:0] operation,  // Operation select: 00=Add, 01=Multiply, 10=Divide
    output reg [15:0] result, // Result: can be up to 16 bits for multiplication
    output reg error         // Error flag for division by zero
);
    always @(*) begin
        error = 0; // Reset error flag
        case (operation)
            2'b00: result = A + B;                      // Addition
            2'b01: result = A * B;                      // Multiplication
            2'b10: begin                                // Division
                if (B == 0) begin
                    error = 1;                          // Division by zero error
                    result = 16'hFFFF;                  // Set result to max value
                end else begin
                    result = A / B;                     // Division
                end
            end
            default: begin
                result = 16'h0000;                      // Default case
                error = 0;
            end
        endcase
    end
endmodule
```

OUTPUT



TESTBENCH CODE

C:/intelFPGA/18.1/AMRUTA/8BITAMDTB.V (/Testbench)

File Edit View Tools Bookmarks Window Help

C:/intelFPGA/18.1/AMRUTA/8BITAMDTB.V (/Testbench) - Default

```
module Testbench;
    reg [7:0] A, B;
    reg [1:0] operation;
    wire [15:0] result;
    wire error;

    // Instantiate the ArithmeticOperations module
    ArithmeticOperations uut (
        .A(A),
        .B(B),
        .operation(operation),
        .result(result),
        .error(error)
    );

    initial begin
        // Test addition
        A = 8'd10; B = 8'd20; operation = 2'b00;
        #10;
        $display("Addition: %d + %d = %d, Error = %b", A, B, result, error);

        // Test multiplication
        A = 8'd15; B = 8'd10; operation = 2'b01;
        #10;
        $display("Multiplication: %d * %d = %d, Error = %b", A, B, result, error);

        // Test division
        A = 8'd100; B = 8'd5; operation = 2'b10;
        #10;
        $display("Division: %d / %d = %d, Error = %b", A, B, result, error);

        // Test division by zero
        A = 8'd100; B = 8'd0; operation = 2'b10;
        #10;
        $display("Division by Zero: %d / %d = %d, Error = %b", A, B, result, error);

        $finish;
    end
endmodule
```

12. 8-bit Register design

Explanation of the Design

1. **Inputs and Outputs**:
   - **clk**: Clock signal to synchronize the register operation.
   - **reset**: Asynchronous reset to clear the register.
   - **load**: Enables loading new data into the register.
   - **data_in**: The 8-bit input data to be loaded.
   - **data_out**: The 8-bit output data of the register.
2. **Behavior**:
   - When `reset` is high, the register is cleared (set to `0`).
   - When `load` is high and `reset` is low, the value of `data_in` is loaded into the register.
   - If neither `reset` nor `load` is asserted, the register holds its previous value.
3. **Testbench**:
   - Tests the register functionality, including reset, load, and hold operations.
   - Generates a clock signal and displays the state of the register during simulation.

MAIN FUNCTION CODE

C:/intelFPGA/18.1/AMRUTA/8BITREG.V

File Edit View Tools Bookmarks Window Help

C:/intelFPGA/18.1/AMRUTA/8BITREG.V - Default

```
module Register8Bit (
    input clk,              // Clock signal
    input reset,            // Asynchronous reset signal
    input load,             // Load enable signal
    input [7:0] data_in,    // 8-bit input data
    output reg [7:0] data_out // 8-bit output data
);

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            data_out <= 8'b0;  // Clear the register on reset
        end else if (load) begin
            data_out <= data_in;  // Load input data into the register
        end
        // If neither reset nor load, data_out holds its value
    end
endmodule
```

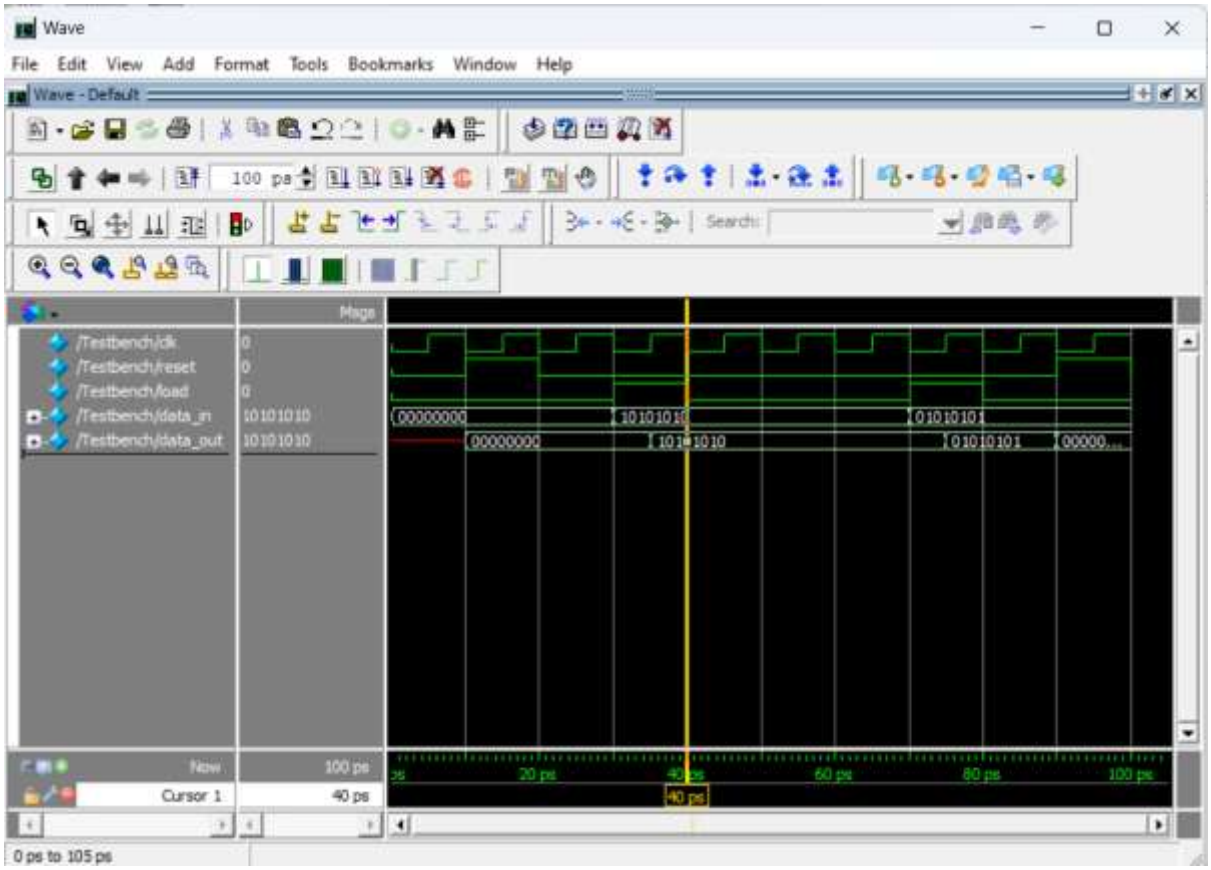
TESTBENCH CODE

File Edit View Tools Bookmarks Window Help

C:/intelFPGA/18.1/AMRUTA/8BITREGTB.V (/Testbench) - Default

```
module Testbench;

    reg clk;
    reg reset;
    reg load;
    reg [7:0] data_in;
    wire [7:0] data_out;

    // Instantiate the 8-bit register
    Register8Bit uut (
        .clk(clk),
        .reset(reset),
        .load(load),
        .data_in(data_in),
        .data_out(data_out)
    );

    // Generate a clock signal
    always #5 clk = ~clk;

    initial begin
        // Initialize signals
        clk = 0; reset = 0; load = 0; data_in = 8'b0;

        // Reset the register
        #10 reset = 1;
        #10 reset = 0;

        // Load a value into the register
        #10 load = 1; data_in = 8'b10101010;
        #10 load = 0;

        // Hold the value
        #20;

        // Load another value into the register
        #10 load = 1; data_in = 8'b01010101;
        #10 load = 0;

        // Reset the register again
        #10 reset = 1;
        #10 reset = 0;

        $finish;
    end

    initial begin
        $monitor("Time: %0t | Reset: %b | Load: %b | Data In: %b | Data Out: %b",
                 $time, reset, load, data_in, data_out);
    end
endmodule
```

OUTPUT



13. Memory unit design and perform memory operations.

1. Memory Unit Design

Components of a Memory Unit

1. **Memory Cells**: Smallest unit that stores a single bit (0 or 1).
2. **Address Decoder**: Decodes the input address to access a specific memory cell or group of cells.
3. **Data Bus**: Transfers data to and from the memory.
4. **Control Signals**:
   - **Read/Write Enable (R/W)**: Determines whether to read from or write to memory.
   - **Chip Select (CS)**: Activates the memory chip for operations.

Memory Array

- A grid of memory cells organized into rows and columns.
- Each cell corresponds to an address, accessed via row and column decoders.

Design Parameters

- **Size**: Total storage capacity (e.g., 64 KB, 1 MB).
- **Word Size**: Number of bits per word (e.g., 8-bit, 16-bit).
- **Access Time**: Time to perform a read/write operation.

Block Diagram

A basic block diagram includes:

- **Address Lines**: For specifying memory location.
- **Data Lines**: For data transfer.
- **Control Lines**: For enabling read/write and chip operations.

2. Perform Memory Operations

Read Operation

1. **Input Address**: Provide the memory location to the address lines.
2. **Enable Chip**: Activate the chip using the Chip Select signal.
3. **Set to Read Mode**: Set R/W signal to 'Read'.
4. **Data Retrieval**: Data from the specified address is sent to the data bus.

Write Operation

1. **Input Address**: Specify the memory location to be written.
2. **Enable Chip**: Activate the chip using the Chip Select signal.
3. **Set to Write Mode**: Set R/W signal to 'Write'.
4. **Data Input**: Send the data to the data bus to write into the specified address.

Example Design: 4x4 Memory Unit

A 4x4 memory unit has:

- **4 rows and 4 columns**: 16 memory cells.
- **Address Lines**: 4-bit address (to represent 16 locations).
- **Data Lines**: 1-bit data bus (single-bit data per operation).

Steps for Read/Write Operation:

1. **Decode Address**: Use the 4-bit address to locate the specific memory cell.
2. **Activate Control Signals**: Depending on the operation (Read/Write), set R/W.
3. **Data Transfer**: Perform the operation through the data bus.

MAIN FUNCTION CODE

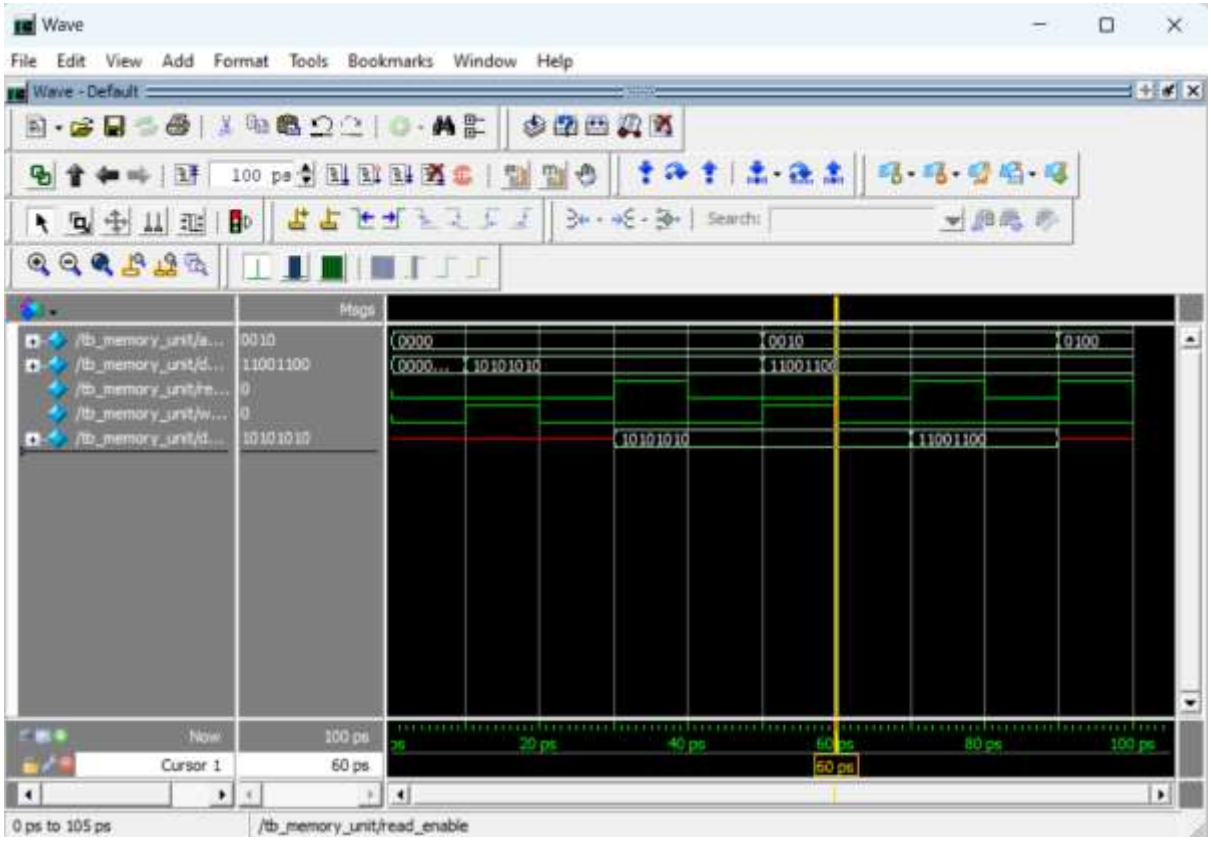
C:/intelFPGA/18.1/AMRUTA/44MU.V

File Edit View Tools Bookmarks Window Help

C:/intelFPGA/18.1/AMRUTA/44MU.V - Default

```
module memory_unit (
    input wire [3:0] address,
    input wire [7:0] data_in,
    input wire read_enable,
    input wire write_enable,
    output reg [7:0] data_out
);
    reg [7:0] memory [0:15]; // 4x4 memory array

    always @(*) begin
        if (read_enable) begin
            data_out = memory[address]; // Read operation
        end else if (write_enable) begin
            memory[address] = data_in; // Write operation
        end
    end
endmodule
```

OUTPUT



TESTBENCH CODE

C:/intelFPGA/18.1/AMRUTA/44MUTB.V (/tb_memory_unit)

File Edit View Tools Bookmarks Window Help

C:/intelFPGA/18.1/AMRUTA/44MUTB.V (/tb_memory_unit) - Default

```
module tb_memory_unit;
    // Testbench signals
    reg [3:0] address;
    reg [7:0] data_in;
    reg read_enable;
    reg write_enable;
    wire [7:0] data_out;

    // Instantiate the memory unit
    memory_unit uut (
        .address(address),
        .data_in(data_in),
        .read_enable(read_enable),
        .write_enable(write_enable),
        .data_out(data_out)
    );

    // Testbench logic
    initial begin
        // Initialize signals
        address = 4'b0000;
        data_in = 8'b00000000;
        read_enable = 0;
        write_enable = 0;

        // Monitor the signals for debugging
        $monitor("Time: %0d | Addr: %b | Data In: %b | Read: %b | Write: %b | Data Out: %b",
                 $time, address, data_in, read_enable, write_enable, data_out);

        // Test Case 1: Write data to address 0000
        #10 address = 4'b0000;
        data_in = 8'b10101010;
        write_enable = 1;
        #10 write_enable = 0;

        // Test Case 2: Read data from address 0000
        #10 read_enable = 1;
        #10 read_enable = 0;

        // Test Case 3: Write data to address 0010
        #10 address = 4'b0010;
        data_in = 8'b11001100;
        write_enable = 1;
        #10 write_enable = 0;

        // Test Case 4: Read data from address 0010
        #10 read_enable = 1;
        #10 read_enable = 0;

        // Test Case 5: Read data from an uninitialized address
        #10 address = 4'b0100;
        read_enable = 1;
        #10 read_enable = 0;

        // Finish simulation
        #10 $finish;
    end
endmodule
```

14. 8-bit simple ALU design

Specifications:

1. **Input Data:**
   - Two 8-bit inputs: `A` and `B`.
   - A control signal `Opcode` (e.g., 4 bits) to select the operation.
2. **Output Data:**
   - An 8-bit result `Result`.
   - Flags:
     - **Carry (C)**: Indicates carry out from the most significant bit.
     - **Zero (Z)**: Set if the result is 0.
     - **Overflow (V)**: Indicates signed arithmetic overflow.
     - **Negative (N)**: Set if the result is negative (most significant bit is 1 in 2's complement).
3. **Operations (Examples):**
   - Arithmetic: Addition, Subtraction, Increment, Decrement.
   - Logic: AND, OR, XOR, NOT.
   - Shift: Left Shift, Right Shift.
   - Comparison: Equality, Greater Than, Less Than.

Architecture Design:

1. Input Multiplexers (Operation Selection)

Use multiplexers to choose the operation based on the `Opcode`. For example:

- Opcode `0000`: Addition
- Opcode `0001`: Subtraction
- Opcode `0010`: AND
- Opcode `0011`: OR
- ...

2. Arithmetic Unit

- Use a full-adder circuit to perform 8-bit addition.
- Subtraction is achieved using 2's complement (invert `B` and add 1).

3. Logic Unit

- Implement basic bitwise operations (AND, OR, XOR) with simple gates.

4. Shifter

- Use barrel shifters or simple combinational logic for left/right shifting.

5. Comparison Logic

- Compare `A` and `B` by checking bit-by-bit or using subtraction.

6. Flag Logic

- **Carry**: From the most significant full-adder.
- **Zero**: All bits of `Result` are 0.
- **Overflow**: Detect using signed addition/subtraction rules.
- **Negative**: Most significant bit of `Result`.

Circuit Design:

Here's an example design flow:

1. **Input Interface**:
   - Two 8-bit registers `A` and `B`.
   - A 4-bit Opcode.
2. **Operational Blocks**:
   - **Arithmetic Block**: Implement an 8-bit ripple-carry adder/subtractor.
   - **Logic Block**: Perform bitwise operations using logic gates.
   - **Shifting Block**: Use combinational shift circuits.
   - **Comparison Block**: Subtract `A` and `B` to derive comparison outputs.

3. **Control Logic:**
   - Decodes the Opcode and enables the corresponding operational block.
4. **Output Multiplexer:**
   - Selects the output of the appropriate operational block.
5. **Flags Generator:**
   - Logic to generate Carry, Zero, Overflow, and Negative flags.

MAIN FUNCTION CODE

C:/intelFPGA/18.1/AMRUTA/8BITALU.V - Default

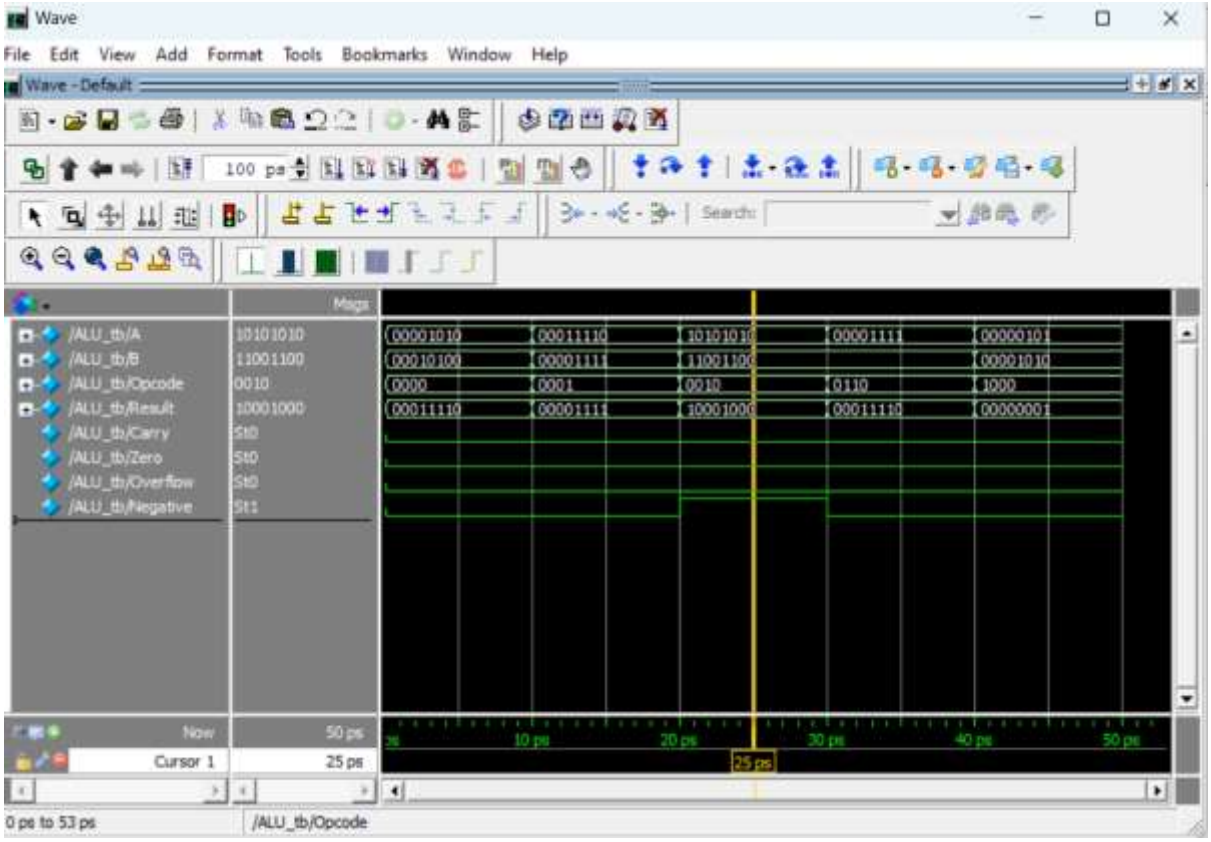
```
module ALU(
    input [7:0] A, B,  // 8-bit inputs
    input [3:0] Opcode, // 4-bit Opcode
    output reg [7:0] Result, // 8-bit Result
    output reg Carry, Zero, Overflow, Negative // Flags
);

    always @(*) begin
        // Default values for flags
        Carry = 0;
        Zero = 0;
        Overflow = 0;
        Negative = 0;

        case (Opcode)
            4'b0000: begin // Addition
                {Carry, Result} = A + B;
                Overflow = (A[7] == B[7]) && (Result[7] != A[7]);
            end
            4'b0001: begin // Subtraction
                {Carry, Result} = A - B;
                Overflow = (A[7] != B[7]) && (Result[7] != A[7]);
            end
            4'b0010: Result = A & B; // Bitwise AND
            4'b0011: Result = A | B; // Bitwise OR
            4'b0100: Result = A ^ B; // Bitwise XOR
            4'b0101: Result = ~A;    // Bitwise NOT
            4'b0110: Result = A << 1; // Logical shift left
            4'b0111: Result = A >> 1; // Logical shift right
            4'b1000: Result = (A < B) ? 8'b1 : 8'b0; // Comparison (A < B)
            default: Result = 8'b0; // Default case
        endcase

        Zero = (Result == 8'b0);
        Negative = Result[7];
    end

endmodule
```

TESTBENCH CODE

C:/intelFPGA/18.1/AMRUTA/8BITALUTB.V (/ALU_tb) - Default

```
module ALU_tb;

    reg [7:0] A, B;         // Inputs to the ALU
    reg [3:0] Opcode;       // ALU operation selector
    wire [7:0] Result;      // ALU output
    wire Carry, Zero, Overflow, Negative; // ALU flags

    // Instantiate the ALU module
    ALU uut (
        .A(A),
        .B(B),
        .Opcode(Opcode),
        .Result(Result),
        .Carry(Carry),
        .Zero(Zero),
        .Overflow(Overflow),
        .Negative(Negative)
    );

    initial begin
        // Test 1: Addition
        A = 8'd10; B = 8'd20; Opcode = 4'b0000;
        #10;

        // Test 2: Subtraction
        A = 8'd30; B = 8'd15; Opcode = 4'b0001;
        #10;

        // Test 3: Bitwise AND
        A = 8'b10101010; B = 8'b11001100; Opcode = 4'b0010;
        #10;

        // Test 4: Logical shift left
        A = 8'b00001111; Opcode = 4'b0110;
        #10;

        // Test 5: Comparison (A < B)
        A = 8'd5; B = 8'd10; Opcode = 4'b1000;
        #10;

        $stop; // Stop simulation
    end

endmodule
```

OUTPUT



Explanation

- **ALU Module**: Implements the functionality described in the image, using a `case` statement to handle operations based on the `Opcode`.
- **Flags**: Carry, Zero, Overflow, and Negative are computed as needed for arithmetic and logic operations.
- **Testbench**: Simulates a few representative cases, including addition, subtraction, logic operations, and comparisons, with delays (`#10`) for observation.

15. 8-bit simple CPU design

Explanation

CPU Design

- **Registers**: A 16-register file is implemented to store data. Registers are addressed using the lower 4 bits of the instruction (Operand).
- **Program Counter**: Keeps track of the current instruction address.
- **Accumulator**: Acts as the central working register for computations.
- **Instruction Set**: Includes basic operations like `LOAD`, `STORE`, `ADD`, `SUB`, bitwise operations (`AND`, `OR`), and branching (`JUMP` and conditional jumps).

Testbench

- Simulates a sequence of instructions, such as loading data into the accumulator, performing arithmetic operations, and branching.
- A simple clock is generated using the `always` block.
- `#10` delays are added to observe the results of each instruction execution.

MAIN FUNCTION CODE

C:/intelFPGA/18.1/AMRUTA/8BITCPU.V - Default

```
module SimpleCPU(
    input clk, reset,
    input [7:0] instruction, // 8-bit instruction: [Opcode (4 bits), Operand (4 bits)]
    output reg [7:0] Accumulator, // 8-bit Accumulator
    output reg [7:0] Address, // 8-bit Address Register
    output reg Zero, Negative // Flags
);

    reg [7:0] RegisterFile [0:15]; // 16 registers (4-bit addressable)
    reg [7:0] ProgramCounter; // 8-bit Program Counter

    wire [3:0] Opcode = instruction[7:4];
    wire [3:0] Operand = instruction[3:0];

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            // Reset all components
            ProgramCounter <= 0;
            Accumulator <= 0;
            Address <= 0;
            Zero <= 0;
            Negative <= 0;
        end else begin
            case (Opcode)
                4'b0000: begin // LOAD: Accumulator <= Register[Operand]
                    Accumulator <= RegisterFile[Operand];
                end
                4'b0001: begin // STORE: Register[Operand] <= Accumulator
                    RegisterFile[Operand] <= Accumulator;
                end
                4'b0010: begin // ADD: Accumulator += Register[Operand]
                    Accumulator <= Accumulator + RegisterFile[Operand];
                    Zero <= (Accumulator == 0);
                    Negative <= Accumulator[7];
```

```
                4'b0011: begin // SUB: Accumulator -= Register[Operand]
                    Accumulator <= Accumulator - RegisterFile[Operand];
                    Zero <= (Accumulator == 0);
                    Negative <= Accumulator[7];
                end
                4'b0100: begin // AND: Accumulator &= Register[Operand]
                    Accumulator <= Accumulator & RegisterFile[Operand];
                    Zero <= (Accumulator == 0);
                    Negative <= Accumulator[7];
                end
                4'b0101: begin // OR: Accumulator |= Register[Operand]
                    Accumulator <= Accumulator | RegisterFile[Operand];
                    Zero <= (Accumulator == 0);
                    Negative <= Accumulator[7];
                end
                4'b0110: begin // JUMP: ProgramCounter <= Operand
                    ProgramCounter <= Operand;
                end
                4'b0111: begin // JUMP IF ZERO: if (Zero) ProgramCounter <= Operand
                    if (Zero) ProgramCounter <= Operand;
                end
                4'b1000: begin // JUMP IF NEGATIVE: if (Negative) ProgramCounter <= Operand
                    if (Negative) ProgramCounter <= Operand;
                end
                default: begin
                    // NOP or undefined instructions
                    Accumulator <= Accumulator;
                end
            endcase

            // Increment Program Counter
            ProgramCounter <= ProgramCounter + 1;
        end
    end
endmodule
```

OUTPUT

TESTBENCH CODE

C:/intelFPGA/18.1/AMRUTA/8BITCPUTB.V - Default

```
module SimpleCPU_tb;

    reg clk, reset;
    reg [7:0] instruction;
    wire [7:0] Accumulator, Address;
    wire Zero, Negative;

    // Instantiate the SimpleCPU
    SimpleCPU uut (
        .clk(clk),
        .reset(reset),
        .instruction(instruction),
        .Accumulator(Accumulator),
        .Address(Address),
        .Zero(Zero),
        .Negative(Negative)
    );

    // Clock generation
    always #5 clk = ~clk;

    initial begin
        // Initialize inputs
        clk = 0;
        reset = 1;
        instruction = 8'b00000000; // NOP

        // Apply reset
        #10 reset = 0;

        // Test 1: LOAD R1 into Accumulator
        #10 instruction = 8'b00000001; // LOAD R1 (register 1)

        // Test 2: ADD R2 to Accumulator
        #10 instruction = 8'b00100010; // ADD R2 (register 2)

        // Test 3: STORE Accumulator into R3
        #10 instruction = 8'b00010011; // STORE R3 (register 3)

        // Test 4: JUMP to Address 5
        #10 instruction = 8'b01100101; // JUMP to address 5

        // Test 5: SUB R4 from Accumulator
        #10 instruction = 8'b00110100; // SUB R4 (register 4)

        // Stop the simulation
        #50 $stop;
    end
endmodule
```

Overview

This module represents a simple 8-bit CPU with:

- **Inputs**:
  - `clk`: The clock signal, which synchronizes operations.
  - `reset`: Resets the CPU to its initial state.
  - `instruction`: An 8-bit instruction consisting of a 4-bit **Opcode** and a 4-bit **Operand**.
- **Outputs**:
  - `Accumulator`: An 8-bit register used to hold the result of computations.
  - `Address`: An 8-bit address register (could be used for memory operations).
  - `Zero` and `Negative` flags: Indicate if the result is zero or negative, respectively.

Flags

- **Zero** (`Zero`):
  - Indicates whether the `Accumulator` value is zero.
- **Negative** (`Negative`):
  - Indicates the sign of the `Accumulator` value (1 for negative, 0 for non-negative).

Flow of Execution

1. On reset:
   - All components are initialized.
   - Execution starts at address 0.
2. On every clock cycle:
   - The `Opcode` is decoded.
   - The corresponding operation is performed using the `Operand` as needed.
   - The `ProgramCounter` is incremented (or updated by a jump instruction).

Applications

- This simple CPU can execute a basic set of operations (arithmetic, logic, load/store, branching).
- It can be extended to include more instructions or features, such as interrupts, memory-mapped I/O, or pipelining.

16. Interfacing of CPU and Memory

Understanding CPU-Memory Interfacing

The CPU and memory communicate via a common interface. The CPU uses this interface to:

1. **Read Data**: Fetch data stored at a specific memory address.
2. **Write Data**: Store data at a specific memory address.

A typical CPU-memory interface involves:

- **Address Bus**: Specifies the memory location (address) being accessed.
- **Data Bus**: Transfers the data between CPU and memory.
- **Control Signals**:
  - **Read Enable** (`read_en`): Indicates a read operation.
  - **Write Enable** (`write_en`): Indicates a write operation.
  - **Clock** (`clk`): Synchronizes the operations.
  - **Reset** (`rst`): Resets the memory or interface logic.

The memory in our case is modeled as a small, simple block of RAM where:

- The **Address Bus** is 8 bits wide, allowing access to 256 locations.
- Each memory location stores 8-bit data.

Step-by-Step Code Explanation

1. CPU-Memory Interface Module

This module defines the core logic for interfacing:

- **Inputs**:
  - `clk`: Synchronizes operations.
  - `rst`: Initializes memory to a default state (all zeros).
  - `read_en`: Signals a read operation.
  - `write_en`: Signals a write operation.
  - `address`: Specifies the memory location to access.
  - `data_in`: Input data to write into memory.
- **Output**:
  - `data_out`: Data fetched from memory during a read operation.

What Happens During Simulation?

1. **Reset Phase**:
   - The memory is initialized to all zeros.
2. **Write Operations**:
   - Data is written to specific addresses. For example:
     - At time `10`: `0xAA` is written to address `10`.
     - At time `20`: `0x55` is written to address `20`.
3. **Read Operations**:
   - Data is read from the same addresses to verify correctness:
     - At address `10`, the value `0xAA` is fetched.
     - At address `20`, the value `0x55` is fetched.

CPU MAIN FUNCTION CODE

File Edit View Tools Bookmarks Window Help

C:/intelFPGA/18.1/AMRUTA/CPUMEMINT.V - Default

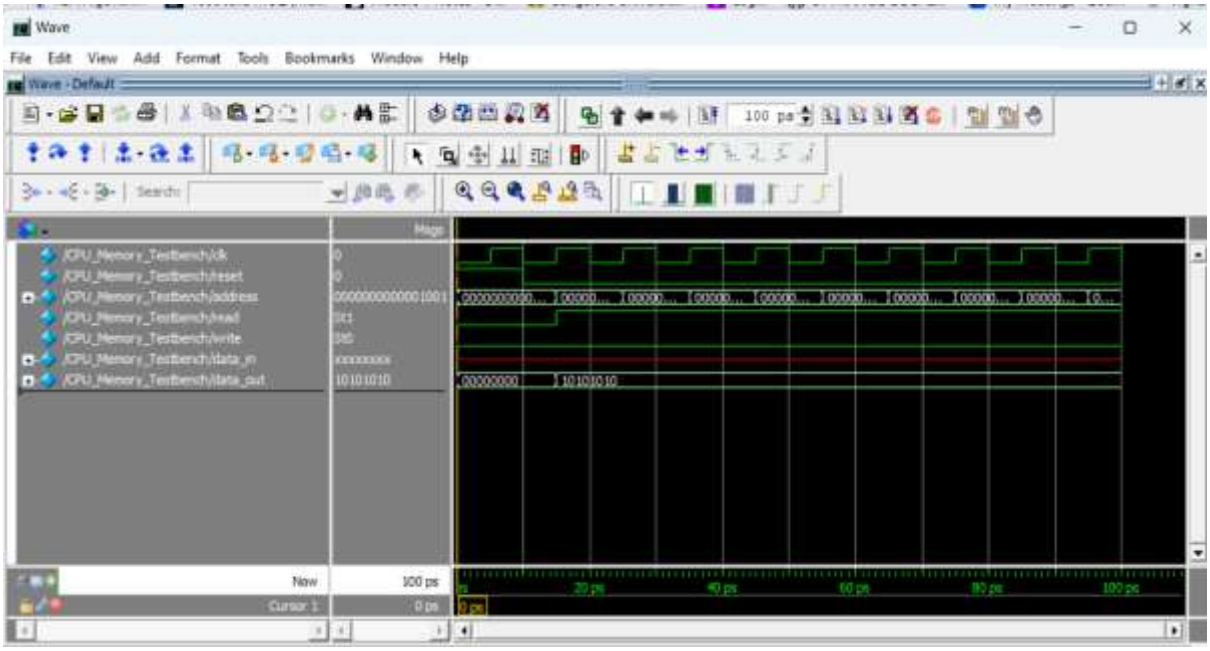
```
module CPU (
    input wire clk,
    input wire reset,
    output reg [15:0] address,
    output reg read,
    output reg write,
    output reg [7:0] data_out,
    input wire [7:0] data_in
);
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            address <= 16'h0000;
            read <= 0;
            write <= 0;
            data_out <= 8'h00;
        end else begin
            // Example behavior
            address <= address + 1; // Increment address
            read <= 1;
            write <= 0; // Read-only for simplicity
            data_out <= 8'hAA; // Placeholder data for write
        end
    end
endmodule
```

MEMORY MODULE CODE

C:/intelFPGA/18.1/AMRUTA/mem.v - Default

```
module Memory (
    input wire clk,
    input wire [15:0] address,
    input wire read,
    input wire write,
    input wire [7:0] data_in,
    output reg [7:0] data_out
);
    reg [7:0] mem [0:65535]; // 64KB memory

    always @(posedge clk) begin
        if (write)
            mem[address] <= data_in; // Write data to memory
        if (read)
            data_out <= mem[address]; // Read data from memory
    end
endmodule
```

OUTPUT



TESTBENCHCODE

C:/intelFPGA/18.1/AMRUTA/CPUMEMINTTB.v - Default

```
module CPU_Memory_Testbench;
    reg clk;
    reg reset;
    wire [15:0] address;
    wire read, write;
    wire [7:0] data_in, data_out;

    // Instantiate CPU and Memory
    CPU cpu_inst (
        .clk(clk),
        .reset(reset),
        .address(address),
        .read(read),
        .write(write),
        .data_out(data_out),
        .data_in(data_in)
    );

    Memory mem_inst (
        .clk(clk),
        .address(address),
        .read(read),
        .write(write),
        .data_in(data_out),
        .data_out(data_in)
    );
```

```
    // Clock generation
    always #5 clk = ~clk; // 10ns clock period

    initial begin
        // Initialize signals
        clk = 0;
        reset = 1;

        // Reset sequence
        #10 reset = 0;

        // Run simulation for a specific duration
        #200 $finish;
    end

    initial begin
        $monitor("Time=%0t | Address=%h | Read=%b | Write=%b | DataIn=%h | DataOut=%h",
                 $time, address, read, write, data_in, data_out);
    end
endmodule
```