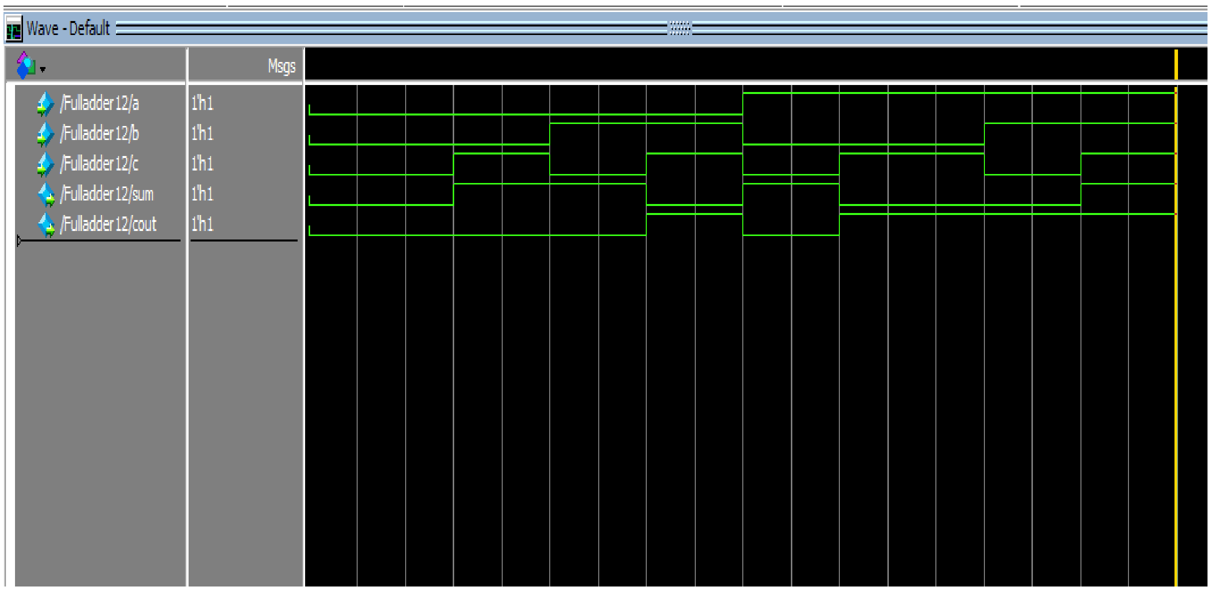
Title: Simulate a Full Adder in VHDL.

Problem Description: Develop a program in Verilog to perform full adder operation.

Full adder

```
module Fulladder12(sum,cout,a,b,c);

input a,b,c;

output sum,cout;

assign sum=a^b^c;

assign cout=(a&b)|(b&c)|(c&a);

endmodule;
```

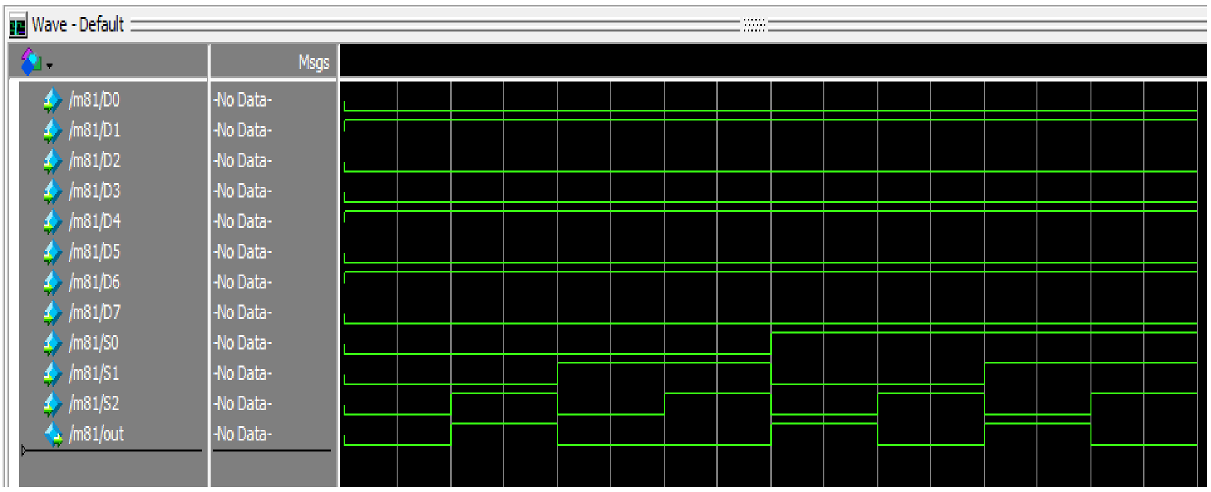


Title: Simulate a 8:1 MUX in VHDL

Problem Description: Develop a program in VHDL to perform 8:1 multiplexer operation.

```
module m81(out, D0, D1, D2, D3, D4, D5, D6, D7, S0, S1, S2);

input wire D0, D1, D2, D3, D4, D5, D6, D7, S0, S1, S2;

output reg out;

always@(*)

begin

case({S0,S1,S2})

3'b000: out=D0;
```

```
3'b001: out=D1;

3'b010: out=D2;

3'b011: out=D3;

3'b100: out=D4;

3'b101: out=D5;

3'b110: out=D6;

3'b111: out=D7;

default: out=1'b0;

endcase

end

endmodule
```



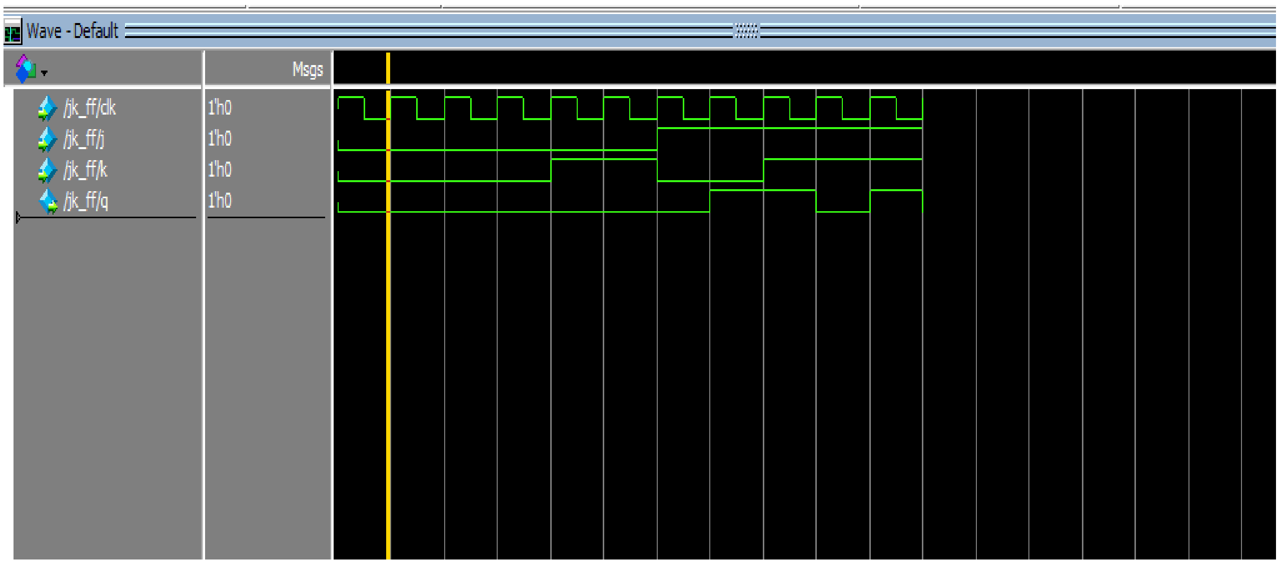
```
/m81/D0   -No Data-
/m81/D1   -No Data-
/m81/D2   -No Data-
/m81/D3   -No Data-
/m81/D4   -No Data-
/m81/D5   -No Data-
/m81/D6   -No Data-
/m81/D7   -No Data-
/m81/S0   -No Data-
/m81/S1   -No Data-
/m81/S2   -No Data-
/m81/out  -No Data-
```

Title: Simulate a Flip-Flops in VHDL.

Problem Description: Develop a program in VHDL to check the working of flip-flops (JK/D/T)

```
module jk_ff ( input j, input k, input clk, output reg q);
    initial begin
q=0;
end
```

```
always @ (posedge clk)

begin

    case ({j,k})
      2'b00 :  q <= q;
      2'b01 :  q <= 0;
      2'b10 :  q <= 1;
      2'b11 :  q <= ~q;
    endcase
end

endmodule
```

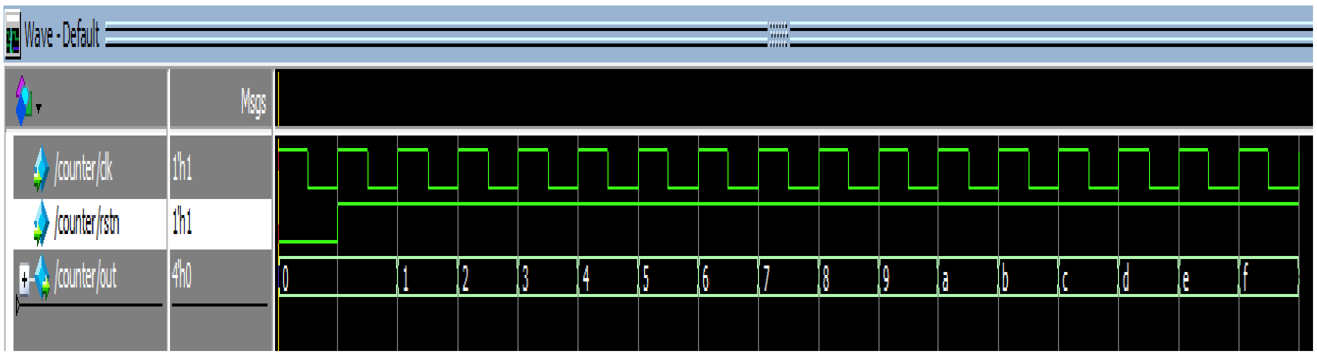


Counter

Title: Simulate a 4-bit binary counter VHDL program.

Problem Description: Develop a program in VHDL to check 4-bit binary counter.

```
module counter (input clk,      // Declare input port for the clock to allow counter to count up
                input rstn,            // Declare input port for the reset to allow the counter to be reset to 0
when required
                output reg[3:0] out);    // Declare 4-bit output port to get the counter values


 // This always block will be triggered at the rising edge of clk (0->1)
  // Once inside this block, it checks if the reset is 0, then change out to zero
  // If reset is 1, then the design should be allowed to count up, so increment the counter


  always @ (posedge clk) begin
    if (! rstn)
      out <= 0;
    else
      out <= out + 1;
  end
endmodule
```



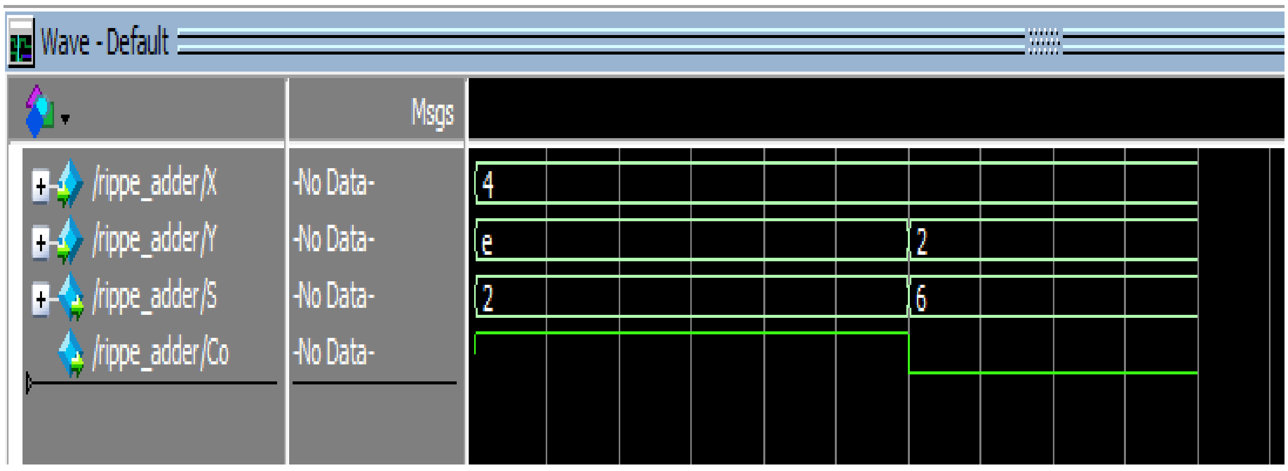
Ripple carry adder

Title: Simulate a 4-bit binary counter VHDL program.

Problem Description: Develop a program in VHDL to check 4-bit binary counter.

```
module rippe_adder(X, Y, S, Co);
```

```
 input [3:0] X, Y;// Two 4-bit inputs
 output [3:0] S;
 output Co;
 wire w1, w2, w3;
 // instantiating 4 1-bit full adders in Verilog
 fulladder u1(X[0], Y[0], 1'b0, S[0], w1);
 fulladder u2(X[1], Y[1], w1, S[1], w2);
 fulladder u3(X[2], Y[2], w2, S[2], w3);
 fulladder u4(X[3], Y[3], w3, S[3], Co);
endmodule

module Fulladder12(a,b,c,sum,cout,);
input a,b,c;
output sum,cout;
assign sum=a^b^c;
assign cout=(a&b)|(b&c)|(c&a);
endmodule;
```

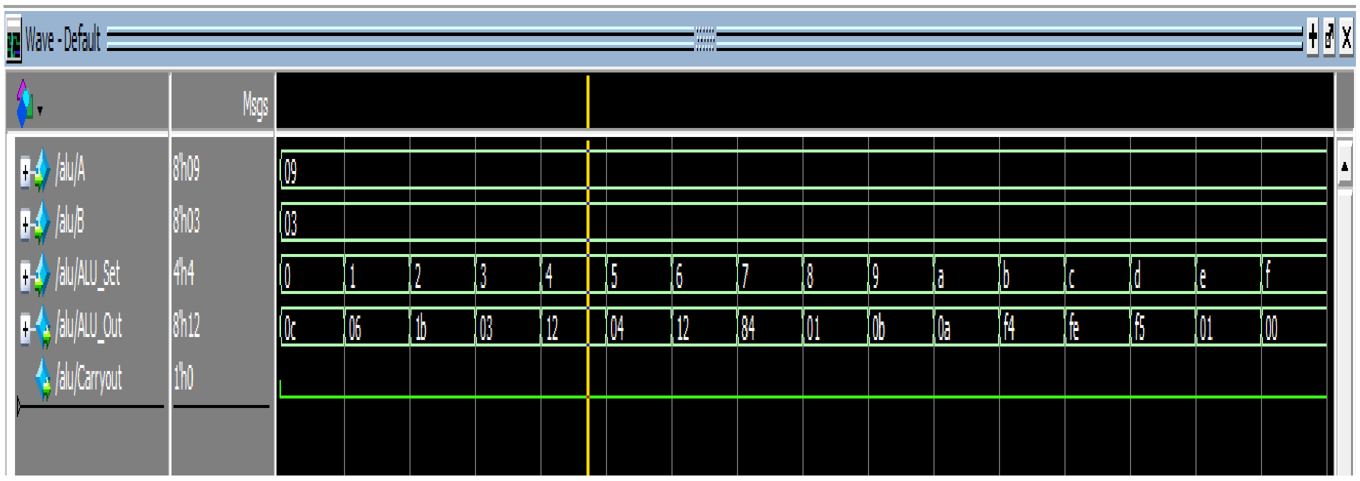


4 bit ALU

Title: Design of a 4-bit ALU.

Problem Description: Develop a program in VHDL to perform 4-bit ALU.

```
module alu(
          input [7:0] A,B,  // ALU 8-bit Inputs
          input [3:0] ALU_Sel,// ALU Selection
          output [7:0] ALU_Out, // ALU 8-bit Output
          output CarryOut // Carry Out Flag
    );
    reg [7:0] ALU_Result;
    wire [8:0] tmp;
    assign ALU_Out = ALU_Result; // ALU out
    assign tmp = {1'b0,A} + {1'b0,B};
    assign CarryOut = tmp[8]; // Carryout flag
    always @(*)
    begin
        case(ALU_Sel)
        4'b0000: // Addition
          ALU_Result = A + B ;
        4'b0001: // Subtraction
          ALU_Result = A - B ;
        4'b0010: // Multiplication
          ALU_Result = A * B;
        4'b0011: // Division
          ALU_Result = A/B;
        4'b0100: // Logical shift left
          ALU_Result = A<<1;
         4'b0101: // Logical shift right
          ALU_Result = A>>1;
         4'b0110: // Rotate left
          ALU_Result = {A[6:0],A[7]};
         4'b0111: // Rotate right
          ALU_Result = {A[0],A[7:1]};
          4'b1000: //  Logical and
          ALU_Result = A & B;
         4'b1001: //  Logical or
          ALU_Result = A | B;
         4'b1010: //  Logical xor
          ALU_Result = A ^ B;
         4'b1011: //  Logical nor
          ALU_Result = ~(A | B);
         4'b1100: // Logical nand
          ALU_Result = ~(A & B);
         4'b1101: // Logical xnor
          ALU_Result = ~(A ^ B);
         4'b1110: // Greater comparison
          ALU_Result = (A>B)?8'd1:8'd0 ;
         4'b1111: // Equal comparison
           ALU_Result = (A==B)?8'd1:8'd0 ;
         default: ALU_Result = A + B ;
        endcase
    end

endmodule
```


Wave - Default
Msgs
/alu/A 8'h09 09
/alu/B 8'h03 03
/alu/ALU_Sel 4'h4 0 1 2 3 4 5 6 7 8 9 a b c d e f
/alu/ALU_Out 8'h12 0c 06 1b 03 12 04 12 84 01 0b 0a f4 fe f5 01 00
/alu/CarryOut 1'h0

INPUT: A-1001(9),B-0011(3)

Alu_SET	Operation	ALU_Result(OUTPUT)
4'b0000: // Addition	A+B	0C
4'b0001: // Subtraction	A-B	06
4'b0010: // Multiplication	A * B	1B
4'b0011: // Division	A/B;	3
4'b0100: // Logical shift left	A<<1;	12(00010010)
4'b0101: // Logical shift right	A>>1	4(00000100)
4'b0110: // Rotate left	{A[6:0],A[7]}	12(00010010)
4'b0111: // Rotate right	{A[0],A[7:1]};	84(10000100)
4'b1000: // Logical and	A & B	01
4'b1001: // Logical or	A | B	0B
4'b1010: // Logical xor	A^B	0A
4'b1011: // Logical nor	~(A | B)	F4
4'b1100: // Logical nand	~(A & B)	FE
4'b1101: // Logical xnor	~(A ^ B)	F5
4'b1110: // Greater comparison	(A>B)	01
4'b1111: // Equal comparison	(A==B)	00