**Ripple carry adder:**  
module ripple\_carry\_add(sum, cout, a, b, cin);

output [3:0] sum;

output cout;

input [3:0] a, b;

input cin;

wire [2:0] c;

module fulladd(sum, cout, a, b, cin);

output sum, cout;

input a, b, cin;

assign sum = a ^ b ^ cin;

assign cout = (a & b) | (cin & (a ^ b));

endmodule

fulladd fa1 (sum[0], c[0], a[0], b[0], cin);

fulladd fa2 (sum[1], c[1], a[1], b[1], c[0]);

fulladd fa3 (sum[2], c[2], a[2], b[2], c[1]);

fulladd fa4 (sum[3], cout, a[3], b[3], c[2]);

endmodule

**4x4 array mul:**

module array\_multiplier\_4x4 (

input [3:0] A, // 4-bit input A

input [3:0] B, // 4-bit input B

output [7:0] P // 8-bit product

);

wire [3:0] PP0, PP1, PP2, PP3;

wire c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11;

wire s1, s2, s3, s4, s5, s6, s7, s8, s9, s10;

assign PP0 = A & {4{B[0]}};

assign PP1 = A & {4{B[1]}};

assign PP2 = A & {4{B[2]}};

assign PP3 = A & {4{B[3]}};

assign P[0] = PP0[0]; // Least significant bit of the product

half\_adder HA1 (PP0[1], PP1[0], P[1], c1);

full\_adder FA1 (PP0[2], PP1[1], c1, s1, c2);

full\_adder FA2 (PP0[3], PP1[2], c2, s2, c3);

half\_adder HA2 (PP1[3], c3, s3, c4);

half\_adder HA3 (s1, PP2[0], P[2], c5);

full\_adder FA3 (s2, PP2[1], c5, s4, c6);

full\_adder FA4 (s3, PP2[2], c6, s5, c7);

full\_adder FA5 (c4, PP2[3], c7, s6, c8);

half\_adder HA4 (s4, PP3[0], P[3], c9);

full\_adder FA6 (s5, PP3[1], c9, s7, c10);

full\_adder FA7 (s6, PP3[2], c10, s8, c11);

full\_adder FA8 (c8, PP3[3], c11, s9, P[7]);

assign P[4] = s7;

assign P[5] = s8;

assign P[6] = s9;

endmodule

module half\_adder (

input A,

input B,

output Sum,

output Carry

);

assign Sum = A ^ B;

assign Carry = A & B;

endmodule

module full\_adder (

input A,

input B,

input Cin,

output Sum,

output Carry

);

assign Sum = A ^ B ^ Cin;

assign Carry = (A & B) | (B & Cin) | (A & Cin);

endmodule

**seven\_seg\_display:**

module sevensigmentDisplay(Seven1, Seven2, Seven3, Seven4);

output reg [6:0] Seven1, Seven2, Seven3, Seven4;

reg [3:0] BCD1, BCD2, BCD3, BCD4;

initial

begin

BCD1=4'b0111;

BCD2=4'b0111;

BCD3=4'b0101;

BCD4=4'b0001;

end

always @(BCD1)

begin

case (BCD1)

4'd0: Seven1 = 7'b1000000;

4'd1: Seven1 = 7'b1111001;

4'd2: Seven1 = 7'b0100100;

4'd3: Seven1 = 7'b0110000;

4'd4: Seven1 = 7'b0011001;

4'd5: Seven1 = 7'b0010010;

4'd6: Seven1 = 7'b0000010;

4'd7: Seven1 = 7'b1111000;

4'd8: Seven1 = 7'b0000000;

4'd9: Seven1 = 7'b0010000;

default: Seven1 = 7'b1111111;

endcase

end

always @(BCD2)

begin

case (BCD2)

4'd0: Seven2 = 7'b1000000;

4'd1: Seven2 = 7'b1111001;

4'd2: Seven2 = 7'b0100100;

4'd3: Seven2 = 7'b0110000;

4'd4: Seven2 = 7'b0011001;

4'd5: Seven2 = 7'b0010010;

4'd6: Seven2 = 7'b0000010;

4'd7: Seven2 = 7'b1111000;

4'd8: Seven2 = 7'b0000000;

4'd9: Seven2 = 7'b0010000;

default: Seven2 = 7'b1111111;

endcase

end

always @(BCD3)

begin

case (BCD3)

4'd0: Seven3 = 7'b1000000;

4'd1: Seven3 = 7'b1111001;

4'd2: Seven3 = 7'b0100100;

4'd3: Seven3 = 7'b0110000;

4'd4: Seven3 = 7'b0011001;

4'd5: Seven3 = 7'b0010010;

4'd6: Seven3 = 7'b0000010;

4'd7: Seven3 = 7'b1111000;

4'd8: Seven3 = 7'b0000000;

4'd9: Seven3 = 7'b0010000;

default: Seven3 = 7'b1111111;

endcase

end

always @(BCD4)

begin

case (BCD4)

4'd0: Seven4 = 7'b1000000;

4'd1: Seven4 = 7'b1111001;

4'd2: Seven4 = 7'b0100100;

4'd3: Seven4 = 7'b0110000;

4'd4: Seven4 = 7'b0011001;

4'd5: Seven4 = 7'b0010010;

4'd6: Seven4 = 7'b0000010;

4'd7: Seven4 = 7'b1111000;

4'd8: Seven4 = 7'b0000000;

4'd9: Seven4 = 7'b0010000;

default: Seven4 = 7'b1111111;

endcase

end

endmodule

**7seg display:**module sevseg(

input wire clk,

output reg [6:0] HEX0,

output reg [6:0] HEX1,

output reg [6:0] HEX2,

output reg [6:0] HEX3,

output reg [6:0] HEX4,

output reg [6:0] HEX5

);

integer i=0,j=0;

// 7-segment encoding (Common Anode: Active LOW)

function [6:0] seg\_decode;

input [3:0] digit;

case (digit)

4'h0: seg\_decode = 7'b1000000;

4'h1: seg\_decode = 7'b1111001;

4'h2: seg\_decode = 7'b0100100;

4'h3: seg\_decode = 7'b0110000;

4'h4: seg\_decode = 7'b0011001;

4'h5: seg\_decode = 7'b0010010;

4'h6: seg\_decode = 7'b0000010;

4'h7: seg\_decode = 7'b1111000;

4'h8: seg\_decode = 7'b0000000;

4'h9: seg\_decode = 7'b0010000;

4'hE: seg\_decode = 7'b0000110;

4'hC: seg\_decode = 7'b1000110;

default: seg\_decode = 7'b1111111;

endcase

endfunction

always@(posedge clk) begin

if(i==0)begin

HEX0 = seg\_decode(4'h7);

HEX1 = seg\_decode(4'h8);

HEX2 = seg\_decode(4'h0);

HEX3 = seg\_decode(4'h1);

HEX4 = seg\_decode(4'hC);

HEX5 = seg\_decode(4'hE);

end

if(i==1)begin

HEX0 = seg\_decode(4'h5);

HEX1 = seg\_decode(4'h0);

HEX2 = seg\_decode(4'h2);

HEX3 = seg\_decode(4'h1);

HEX4 = seg\_decode(4'hC);

HEX5 = seg\_decode(4'hE);

end

j=j+1;

i=j%2;

end

endmodule

**Single Port RAM:**

module singpram(  
input [2:0] data,  
input [5:0] addr,  
input we, clk,  
output [2:0] q  
);  
  
reg [2:0] ram[63:0];  
reg [5:0] addr\_reg;  
  
always @ (posedge clk)begin  
// Write  
if (we)  
ram[addr] <= data;  
addr\_reg <= addr;  
end  
  
assign q = ram[addr\_reg];  
  
endmodule

**Single Port ROM:**module singpram(  
input [5:0] addr,  
input clk,  
output [3:0] q  
);  
reg [3:0] rom[63:0];  
reg [5:0] addr\_reg;  
  
initial begin  
rom[0]=4'b0000;  
rom[1]=4'b0001;  
rom[2]=4'b0010;  
rom[3]=4'b0011;  
rom[4]=4'b0100;  
rom[5]=4'b0101;  
rom[6]=4'b0110;  
rom[7]=4'b0111;  
rom[8]=4'b1000;  
rom[9]=4'b1001;  
rom[10]=4'b1010;  
rom[11]=4'b1011;  
rom[12]=4'b1100;  
rom[13]=4'b1101;  
rom[14]=4'b1110;  
rom[15]=4'b1111;  
end  
  
always @ (posedge clk)begin  
addr\_reg <= addr;  
end  
  
assign q = rom[addr\_reg];  
endmodule

**DUAL PORT RAM:**

module Dual\_Port\_RAM (

input clk,

input we\_a, input we\_b,

input [3:0] addr\_a,

input [3:0] addr\_b,

input [7:0] din\_a,

input [7:0] din\_b,

output reg [7:0] dout\_a, output reg [7:0] dout\_b

);

reg [7:0] ram [15:0];

always @(posedge clk) begin

if (we\_a) ram[addr\_a] <= din\_a;

dout\_a <= ram[addr\_a];

if (we\_b) ram[addr\_b] <= din\_b;

dout\_b <= ram[addr\_b];

end

endmodule

`timescale 1ns / 1ps

module Dual\_Port\_RAM\_tb;

reg clk;

reg we\_a, we\_b;

reg [3:0] addr\_a, addr\_b;

reg [7:0] din\_a, din\_b;

wire [7:0] dout\_a, dout\_b;

Dual\_Port\_RAM uut (

.clk(clk),

.we\_a(we\_a),

.we\_b(we\_b),

.addr\_a(addr\_a),

.addr\_b(addr\_b),

.din\_a(din\_a),

.din\_b(din\_b),

.dout\_a(dout\_a),

.dout\_b(dout\_b)

);

always #5 clk = ~clk;

initial begin

clk = 0;

we\_a = 0; we\_b = 0;

addr\_a = 4'b0000; addr\_b = 4'b0000;

din\_a = 8'b00000000; din\_b = 8'b00000000;

#10;

we\_a = 1; addr\_a = 4'b0001; din\_a = 8'b10101010; #10;

we\_a = 0;

addr\_a = 4'b0001; #10;

we\_b = 1; addr\_b = 4'b0010; din\_b = 8'b11001100; #10;

we\_b = 0;

addr\_b = 4'b0010; #10;

we\_a = 1; addr\_a = 4'b0011; din\_a = 8'b11110000; we\_b = 1; addr\_b = 4'b0011; din\_b = 8'b00001111; #10;

we\_a = 0; we\_b = 0;

addr\_a = 4'b0011; addr\_b = 4'b0011; #10;

$stop;

end

endmodule

**Dual port rom:**

module Dual\_Port\_ROM (

input [3:0] addr\_a, addr\_b,

output reg [7:0] dout\_a, dout\_b

);

reg [7:0] rom [0:15];

initial begin

rom[0] = 8'b00000000; rom[1] = 8'b00001111;

rom[2] = 8'b11110000; rom[3] = 8'b10101010;

rom[4] = 8'b01010101; rom[5] = 8'b11001100;

rom[6] = 8'b00110011; rom[7] = 8'b11111111;

rom[8] = 8'b00000001; rom[9] = 8'b00000110;

rom[10] = 8'b11000011; rom[11] = 8'b10100101;

rom[12] = 8'b10011001; rom[13] = 8'b01101101;

rom[14] = 8'b00110101; rom[15] = 8'b01011010;

end

always @(\*) begin

dout\_a = rom[addr\_a];

dout\_b = rom[addr\_b];

end

endmodule

**vending:**

module vending\_machine(

input clk,

input reset,

input [1:0] coin,

output reg chocolate,

output reg [6:0]seven1,seven2,seven3,seven4

);

reg [2:0] state, next\_state;

localparam S0 = 3'b000,

S5 = 3'b001,

S10 = 3'b010,

S15 = 3'b011,

S\_OVER = 3'b100;

always @(posedge clk or posedge reset) begin

if (reset)

state <= S0;

else

state <= next\_state;

end

always @(\*) begin

next\_state = state;

chocolate = 0;

case (state)

S0: begin

if (coin == 2'b01)

next\_state = S5;

else if (coin == 2'b10)

next\_state = S10;

end

S5: begin

if (coin == 2'b01)

next\_state = S10;

else if (coin == 2'b10)

next\_state = S15;

end

S10: begin

if (coin == 2'b01)

next\_state = S15;

else if (coin == 2'b10)

next\_state = S\_OVER;

end

S15: begin

chocolate = 1;

next\_state = S0;

end

S\_OVER: begin

chocolate = 1;

next\_state = S0;

end

endcase

case(chocolate)

0: begin

seven4 = 7'b0111111;

seven3 = 7'b0111111;

seven2 = 7'b0111111;

seven1 = 7'b0111111;

end

1: begin

seven4 = 7'b1000000;

seven3 = 7'b0001100;

seven2 = 7'b0000110;

seven1 = 7'b0101011;

end

endcase

end

endmodule

**Vending:**

module vending\_machine (

input clk,

input reset,

input [1:0] coin,

input [1:0] product,

output reg dispense,

output reg [2:0] change

);

parameter PRICE1 = 3;

parameter PRICE2 = 5;

parameter PRICE3 = 7;

reg [3:0] balance;

always @(posedge clk or posedge reset) begin

if (reset) begin

balance <= 0;

dispense <= 0;

change <= 0;

end

else begin

case (coin)

2'b01: balance <= balance + 1;

2'b10: balance <= balance + 2;

2'b11: balance <= balance + 5;

default: balance <= balance;

endcase

case (product)

2'b01: if (balance >= PRICE1) begin

dispense <= 1;

change <= balance - PRICE1;

balance <= 0;

end

2'b10: if (balance >= PRICE2) begin

dispense <= 1;

change <= balance - PRICE2;

balance <= 0;

end

2'b11: if (balance >= PRICE3) begin

dispense <= 1;

change <= balance - PRICE3;

balance <= 0;

end

default: begin

dispense <= 0;

change <= 0;

end

endcase

end

end

endmodule

**sequence detector:**

module sequence\_dect (

input clk,

input reset,

input seq\_in,

output reg detected

);

parameter S0 = 3'b000,

S1 = 3'b001,

S2 = 3'b010,

S3 = 3'b011,

S4 = 3'b100;

reg [2:0] state, next\_state;

always @(posedge clk or posedge reset) begin

if (reset)

state <= S0;

else

state <= next\_state;

end

always @(\*) begin

next\_state = S0;

detected = 0;

case (state)

S0: begin

if (seq\_in)

next\_state = S1;

else

next\_state = S0;

end

S1: begin

if (seq\_in)

next\_state = S2;

else

next\_state = S0;

end

S2: begin

if (seq\_in)

next\_state = S3;

else

next\_state = S0;

end

S3: begin

if (seq\_in)

next\_state = S4;

else

next\_state = S0;

end

S4: begin

detected = 1;

if (seq\_in)

next\_state = S4;

else

next\_state = S0;

end

default: next\_state = S0;

endcase

end

endmodule

**8-bit alu:**

module alu(input[2:0]in,output reg [8:0]ans);

reg [7:0]a=8'b11111111;

reg [7:0]b=8'b11101101;

always @(\*) begin

case(in)

3'b000:ans=a+b;

3'b001:ans=a-b;

3'b010:ans=a/b;

3'b011:ans=~a;

3'b100:ans=~b;

3'b101:ans=a&b;

3'b110:ans=a|b;

3'b111:ans=a^b;

default: begin

ans=9'b000000000;

carry=1'b0;

end

endcase

end

endmodule