Program:

```
module half_adder(a,b,sum,carry);
input a;
input b;
output sum;
output carry;
reg sum, carry;
always@(a,b)
begin
sum=a^b;
carry=a&b;
end
endmodule
```

Program:

```
module full_adder(a, b, c, sum, carry);
input a,b,c;
output sum,carry;
wire x,y,z;
assign x=a^b;
assign sum=x^c;
assign y=a&b;
assign z=x&c;
assign carry=z|y;
endmodule
```

Program:

```
module decoder_pro(
   input a,
   input b,
   output w,
   output x,
   output y,
   output z
   );
   assign w=(~a)&(~b);
   assign x=(~a)&b;
    assign y=a&(~b);
     assign z=a&b;
endmodule
```

Program:

```
module encoder_pro(
   input a,
   input b,
   input c,
   input d,
   output x,
   output y,
   output z
   );
   assign x=b|d;
   assign y=c|d;
endmodule
```

Program:

```
module alu_pro (
input [31:0],a,b,
output reg count,
input [2:0] opcode,
output reg [31:0] zout, output reg [32:0] temp);
always@ (a, b)
begin
case (opcode)
4'b0000: temp=a+b;
4'b0001:temp=a-b;
4'b0010: temp [31:0]=a|b;
4'b0011: temp [31:0]=a&b;
4'b0100: temp [31:0]=~a;
4'b0101:temp=a[15:0]*b [15:0];
4'b0110: temp [31:0]=~(a&b);
4'b0111: temp [31:0]=a^b;
default: temp=0000000;
endcase
zout=temp [31:0]; count=temp [32];
temp [32]=0;
end
endmodule
```

Program:

```
module multi_pro(
   input I0,
   input I1,
   input I2,
   input I3,
   input S1,
   input S2,
   output Y
   );
   assign Y =(I0&(~S1)&(~S2))|(I1&(~S1)&(S2))|(I2&S1&(~S2))|(I3&S1&S2);
endmodule
```

Program:

```
module demulti_pro(
   input Din,
   input S1,
   input S2,
   output Y0,
   output Y1,
   output Y2,
   output Y3
   );
   assign Y0=Din &(~S1)&(~S2);
   assign Y1=Din & (~S1)&S2;
   assign Y2=Din & S1&(~S2);
   assign Y3=Din & S1 &S2;
endmodule
```

Program:

```
module sr_ff_pro(s,r,clk,rst, q,qb);
input s,r,clk,rst;
output q,qb;
reg q,qb;
reg [1:0]sr;
always@(posedge clk,posedge rst)
begin
sr={s,r};
if(rst==0)
begin
case (sr)
2'd1:q=1'b0;
2'd2:q=1'b1;
2'd3:q=1'b1;
default: begin end
endcase
end
else
begin
q=1'b0;
end
qb=~q;
end
end module
```

Program:

```
module d_ff_pro(d,clk,q);
input d;
input clk;
output q;
reg q
always @(posedge clk)
begin
q <= d;
end
endmodule
```

Program:

```
module t_ff_pro ( input clk, input rstn,
input t, output reg q);
 always @ (posedge clk) begin
 if (!rstn)
 q <= 0;
 else
 if (t)
 q <= ~q;
 else
 q <= q;
 end
endmodule
```

Program:

```
module jk_ff_pro ( input j, input k,
input clk, output q);
 reg q;
 always @ (posedge clk)
 case ({j,k})
 2'b00 : q <= q;
 2'b01 : q <= 0;
 2'b10 : q <= 1;
 2'b11 : q <= ~q;
 endcase
endmodule
```

Verilog code for up counter:

```
module up_counter(input clk, reset,
output[3:0] counter);
reg [3:0] counter_up;
always @(posedge clk or posedge reset)
begin
if(reset)
counter_up <= 4'd0;
else
counter_up <= counter_up + 4'd1;
end
assign counter = counter_up;
endmodule
```

Verilog testbench code for up counter:

```
module upcounter_testbench();
reg clk, reset;
wire [3:0] counter;
up_counter dut(clk, reset, counter);
initial begin
clk=0;
forever #5 clk=~clk;
end
initial begin
reset=1;
#20;
reset=0;
end
endmodule
```

Verilog code for down counter:

```
module down_counter(input clk, reset,
output [3:0] counter );
reg [3:0] counter_down;
always @(posedge clk or posedge reset)
begin
if(reset)
counter_down <= 4'hf;
else
counter_down <= counter_down - 4'd1;
end
assign counter = counter_down;
endmodule
```

Verilog testbench code for down counter:

```
module downcounter_testbench();
reg clk, reset;
wire [3:0] counter;
down_counter dut(clk, reset,
counter);
initial begin
clk=0;
forever #5 clk=~clk;
end
initial begin
reset=1;
#20;
reset=0;
end
endmodule
```

AND Gate

Structural Model:

```
module andstr(x,y,z);
input x,y;
output z;
and g1(z,x,y);
endmodule
```

Data Flow Model:

```
module anddf(x,y,z);
input x,y;
output z;
assign z=(x&y);
endmodule
```

BehaviouralModel:

```
module andbeh(x,y,z);
input x,y;
output z;
reg z;
always @(x,y)
z=x&y;
endmodule
```

VERILOG Program

b) NAND Gate

Structural Model	**Data Flow Model**	**BehaviouralModel**

module nandstr(x,y,z);
input x,y;
output z;
nand g1(z,x,y);
endmodule
	module nanddf(x,y,z);
	input x,y;
	output z;
	assign z= !(x&y);
	endmodule
		module nandbeh(x,y,z);
		input x,y;
		output z; reg z;
		always @(x,y)
		z=!(x&y);
		endmodule

c) OR Gate

Structural Model	**Data Flow Model**	**BehaviouralModel**

module orstr(x,y,z);
input x,y;
output z;
or g1(z,x,y);
endmodule
	module ordf(x,y,z);
	input x,y;
	output z;
	assign z=(x|y);
	endmodule
		module orbeh(x,y,z);
		input x,y;
		output z; reg z;
		always @(x,y)
		z=x|y;
		endmodule

d) NOR Gate

Structural Model	**Data Flow Model**	**BehaviouralModel**

module norstr(x,y,z);
input x,y;
output z;
nor g1(z,x,y);
endmodule
	module nordf(x,y,z);
	input x,y;
	output z;
	assign z= !(x|y);
	endmodule
		module norbeh(x,y,z);
		input x,y;
		output z;
		reg z;
		always @(x,y)
		z=!(x|y);
		endmodule

d) XOR Gate

Structural Model	Data Flow Model	BehaviouralModel

module xorstr(x,y,z);
input x,y;
output z;
xor g1(z,x,y);
endmodule
	module xordf(x,y,z);
	input x,y;
	output z;
	assign z=(x^y);
	endmodule
		module xorbeh(x,y,z);
		input x,y;
		output z;
		reg z;
		always @(x,y)
		z=x^y;
		endmodule

d) XNOR Gate

Structural Model	Data Flow Model	BehaviouralModel

module xnorstr(x,y,z);
input x,y;
output z;
xnor g1(z,x,y);
endmodule
	module xnordf(x,y,z);
	input x,y;
	output z;
	assign z= !(x^y);
	endmodule
		module xnorbeh(x,y,z);
		input x,y;
		output z; reg z;
		always @(x,y)
		z=!(x^y);
		endmodule

d) NOT Gate

Structural Model	Data Flow Model	BehaviouralModel

module notstr(x,z);
input x;
output z;
not g1(z,x);
endmodule
	module notdf(x,z);
	input x;
	output z;
	assign z= !x;
	endmodule
		module notbeh(x,z);
		input x;
		output z; reg z;
		always @(x)
		z=!x;
		endmodule