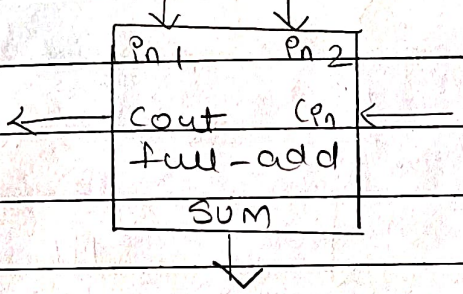
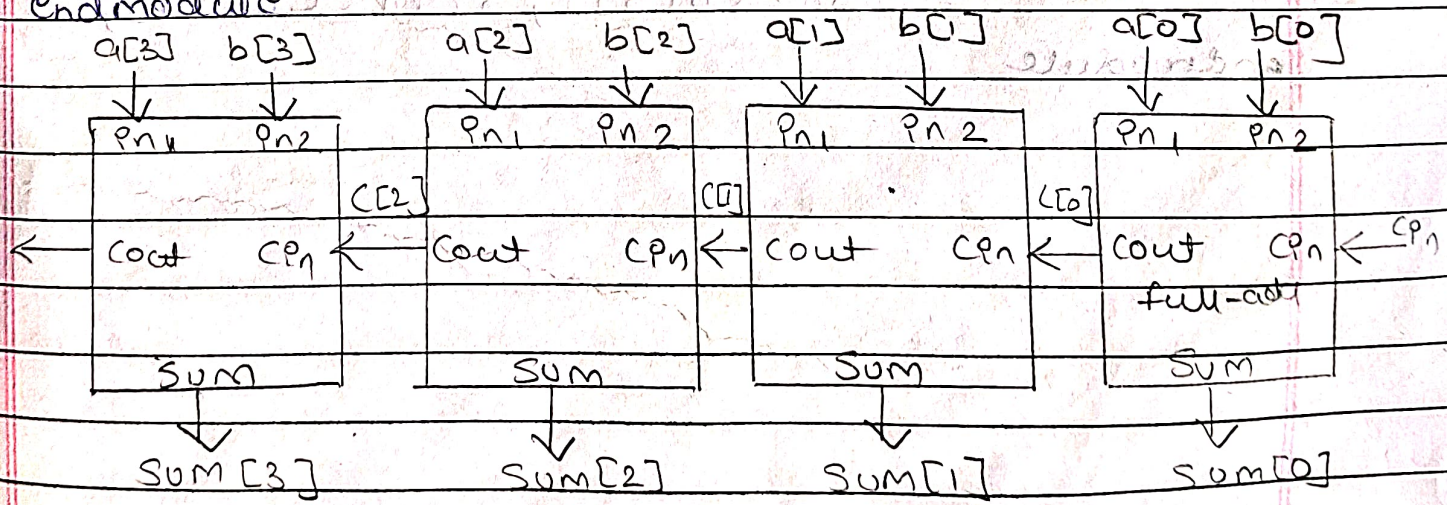
* 4 bit carry ripple adder.

1 bit full adder

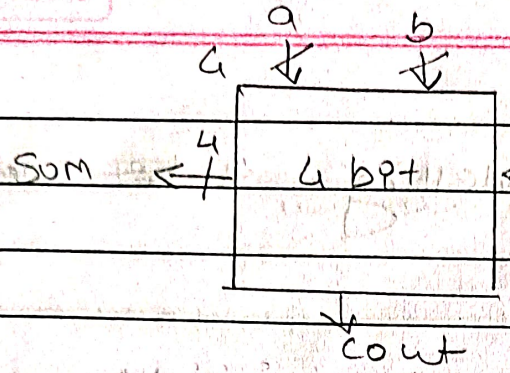


in1 ^ in2 ^ cin
in2 & cin
in1 & cin

Module full-adder (~~Cout~~ cout, Sum, in1, in2, cin);
output cout, Sum;
input in1, in2, cin;
assign Sum = ~~a^b^c;~~ in1 ^ in2 ^ cin;
~~assign cout = ((a&b) | (b&c) | (c&a));~~
assign cout = ((in1 & in2) | (in2 & cin) | (in1 & cin));
endmodule

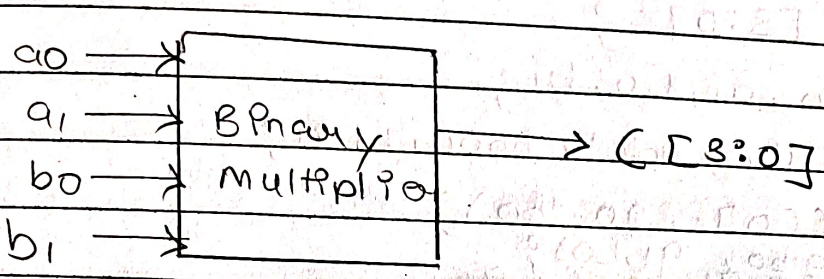


4 bit carry ripple counter

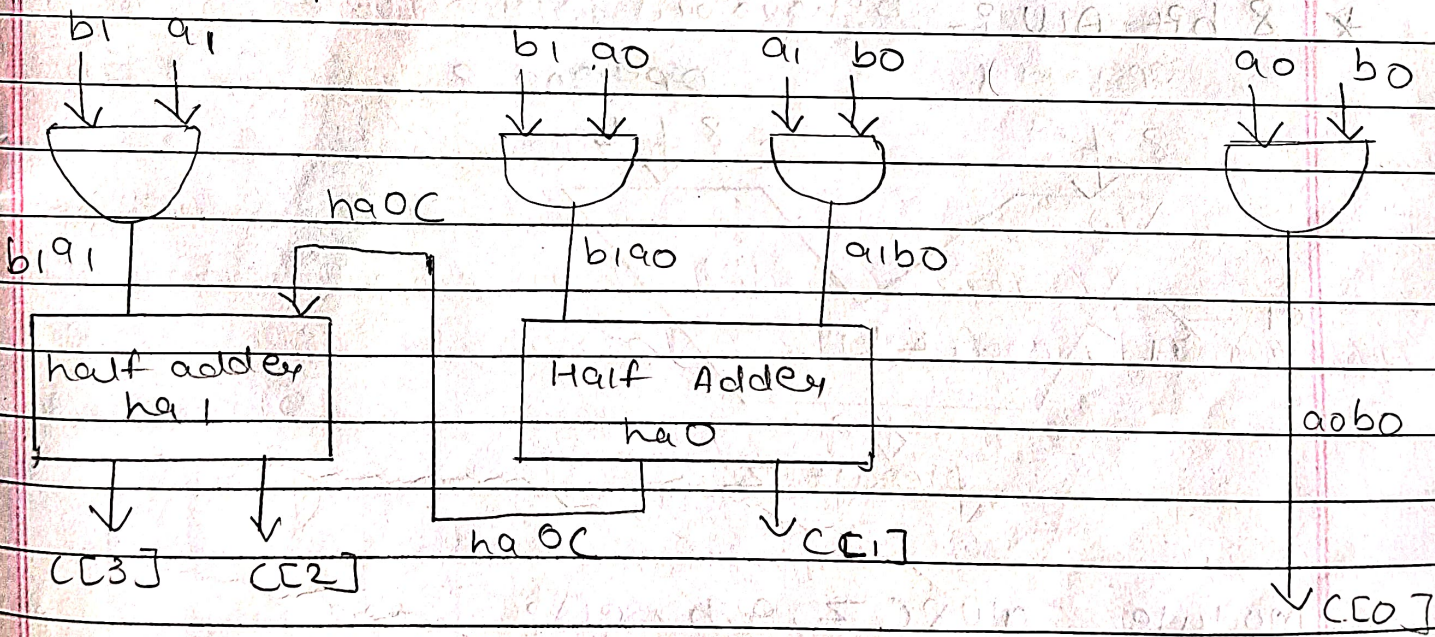



```
module add4bit (cout, sum, a, b, cin);
input [3:0] a, b;
input cin;
output [3:0] sum;
output cout;
wire [2:0] c;
full-add fa1 (.cout(c[0]), .sum(sum[0]), .in1(a[0]
    .in2(b[0]), .cin(cin));
full-add fa2 (.cout(c[1]), .sum(sum[1]), .in1(a[1]
    .in2(b[1]), .cin(c[0]));
full-add fa3 (.cout(c[2]), .sum(sum[2]), .in1(a[
    .in2(b[2]), .cin(c[1]));
full-add fa4 (.cout(cout), .sum(sum[3]), .in1(a[
    .in2(b[3]), .cin(c[2]));
endmodule
```

* Binary multipliers -



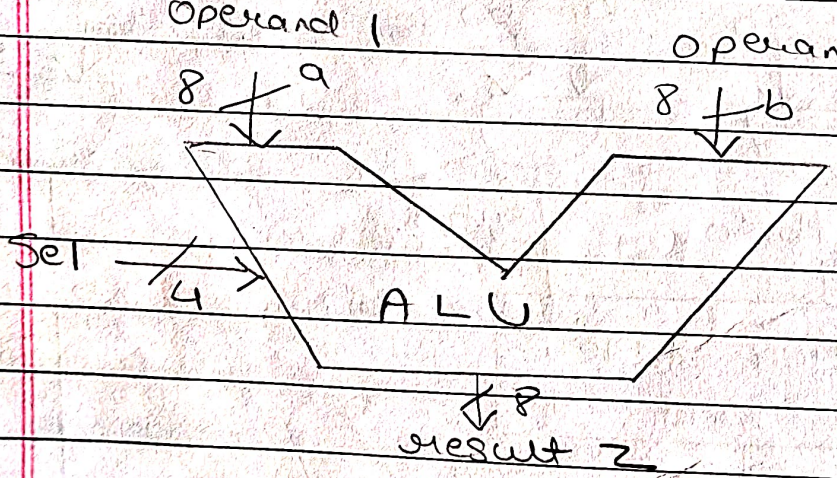
$$
\begin{array}{cc}
a_1 & a_0 \\
b_1 & b_0 \\
\hline
a_1 b_0 & a_0 b_0 \\
b_1 a_1 & b_1 a_0 \quad \times \\
\hline
b_1 a_1 & (a_1 b_0 + b_1 a_0) \; a_0 b_0 \\
C_2 & C_1 \qquad C_0
\end{array}
$$




```
module half-adder(Cout, Sum, a, b);
output Cout, Sum;
input a, b;
assign Sum = a^b;
assign Cout = a & b;
endmodule
```

```
module multiplier(c, a0, a1, b0, b1);
output [3:0] c;
input a0, a1, b0, b1;
wire a1b0, a0b1, haoc, b1a1;
and (c[0], a0, b0);
and (a1b0, a1, b0);
and (b1a0, b1, a0);
and (b1a1, b1, a1);
half_adder ha_0 (c[1], haoc, b1a0, a1b0);
half_adder ha_1 (c[3], c[2], b1a1, haoc);
endmodule
```

* 8 bit ALU :- behavioral Verilog code

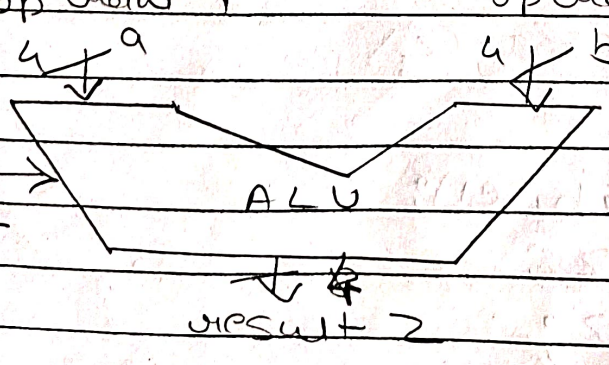



```
Module alu8(z, a, b, sel);
output [7:0] z;
input [7:0] a, b;
input [3:0] sel;
reg [7:0] z;
always@(sel, a, b)
begin
case (sel)
```

```
4'b0000: z = a+b;
4'b0001: z = a-b;
4'b0010: z = b-1;
4'b0011: z = a*b;
4'b0100: z = a&&b;  // logical and
4'b0101: z = a||b;  // logical or
4'b0110: z = !a;    // logical negation
4'b0111: z = ~a;
4'b1000: z = a&b;
4'b1001: z = a|b;
4'b1010: z = a^b;
4'b1011: z = a<<1;  // left shift
4'b1100: z = a>>1;  // right shift
4'b1101: z = a+1;
4'b1110: z = ~(a^b);
4'b1111: z = a-1;
endcase
end
endmodule
```

* Write behavioral verilog code for ALU implementing operation and, or, xor, xnor.

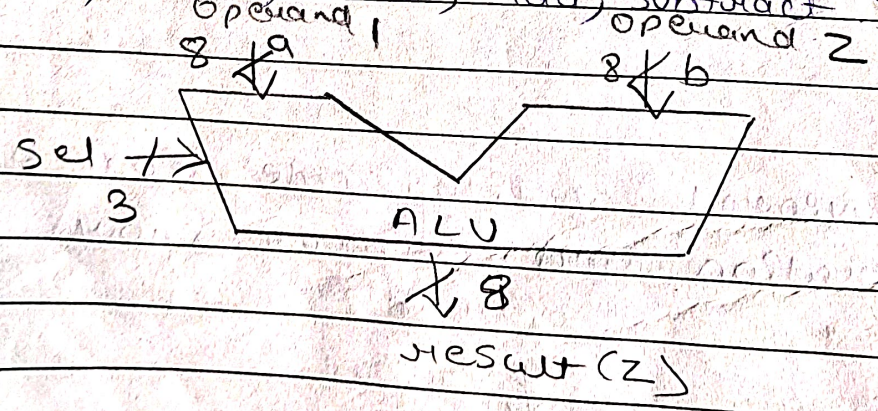
>



```
module alu(z, a, b, sel);
output [3:0] z;
reg [3:0] z;
input [3:0] a, b;
input [1:0] sel;
always@ (sel, a, b)
begin
case(sel)
2'b00 : z = a & b; // And
2'b01 : z = a | b; // OR
2'b10 : z = a ^ b; // xor
2'b11 : z = ~(a ^ b) // xnor
endcase
end
endmodule
```

* Design 8 bit ALU performing and, or, nor, xor, xnor, nand, add, subtract

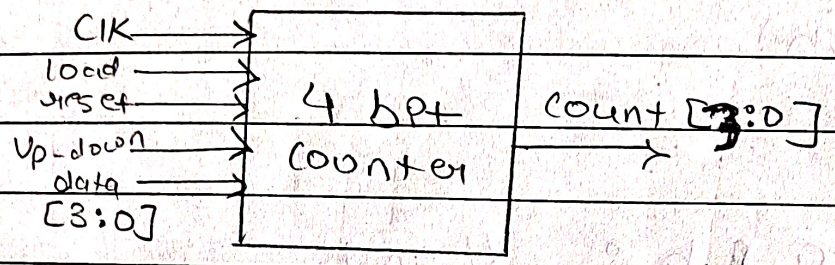
→




```
module alu(z, a, b, sel);
output [7:0] z;
reg [7:0] z;
```

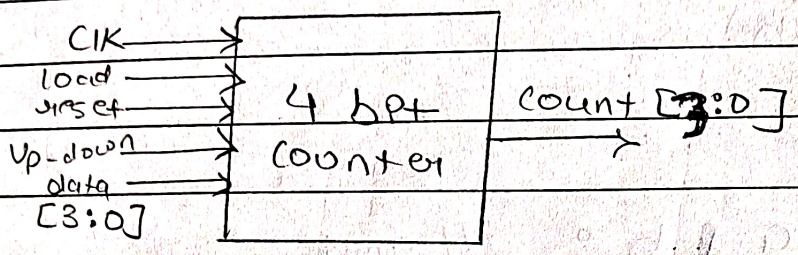
```
input [7:0] a,b;
input [2:0] sel;
always@ (sel,a,b)
begin
 case (sel)
3'b000 : z = a & b;   // and
3'b001 : z = a | b;   // or
3'b010 : z = a ^ b;   // xor
3'b011 : z = ~(a^b);  // xnor
3'b100 : z = ~(a|b);  // nor
3'b101 : z = ~(a&b);  // nand
3'b110 : z = a+b;     // add
3'b111 : z = a-b ;    // sub
endcase
end
endmodule
```

IMP ★ 4 bit counter :-



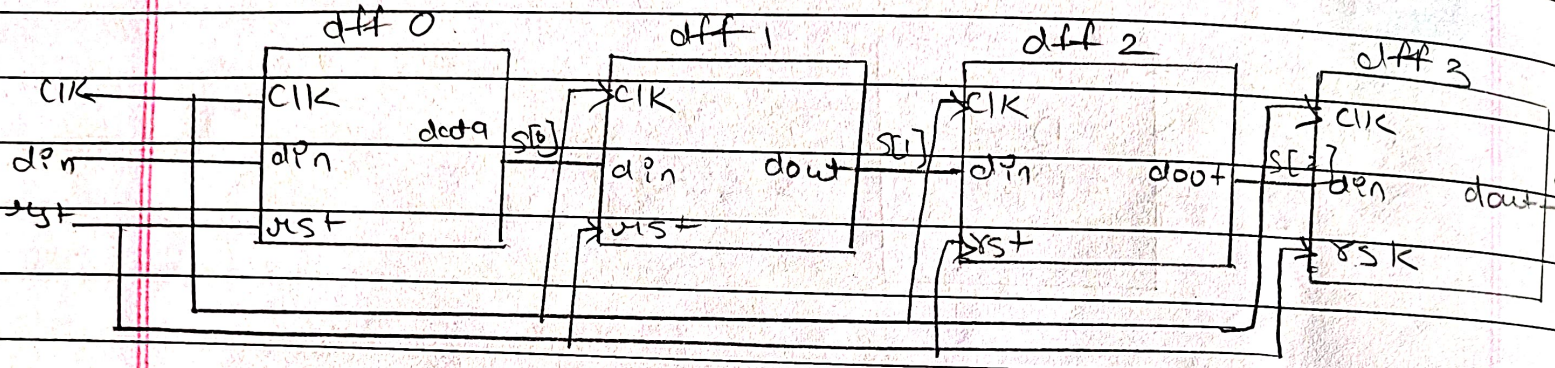

```
Module Counter( count, CIK, reset, load, up-down, dat
output reg [3:0] count;
input CIK, reset, load, up-down;
input [3:0] data;
always@ (posedge CIK)
begin
if (reset) // set count to zero
 count = 0;
else if (load) // load the counter with data
 count = data;
else if (UP_down) // counts up
 count = count + 1;
else
 count = count - 1;
end
endmodule
```

4 bit counter :-




```
Module Counter(count, CIK, reset, load, up-down, data);
output reg [3:0] count;
input CIK, reset, load, up-down;
input [3:0] data;
always@(posedge CIK)
begin
if(reset) // set count to zero
  count = 0;
else if(load) // load the counter with data value
  count = data;
else if(up_down) // counts up
  count = count + 1;
else
  count = count - 1;
end
endmodule
```

* 4 bit Serial In Serial out Shift register:-




```
Module dff (dout, clk, din, rst);
input clk, din, rst;
output reg dout;
always@ (posedge clk)
begin
  if (rst)
    dout = 1;
  else
    dout = din;
end
endmodule

Module sipo (dout, clk, din, rst);
output reg dout;
input clk, din, rst;
wire [2:0] S;
dff dff0 (.dout(S[0]), .clk(clk), .din(din),
          .rst(rst));
dff dff1 (.dout(S[1]), .clk(clk), .din(S[0]),
          .rst(rst));
```

dff dff2 (.dout(S[2]), .clk(clk), .din(S[1]), .
rst(rst));
dff dff3 (.dout(dout), .clk(clk), .din(S[2]),
.rst(rst));
endmodule

TMP