

Operatorer - logik

<i>operator</i>	<i>bitwise</i>	<i>logisk</i>	<i>exempel(bitwise)</i>	<i>exempel(logisk)</i>
AND	&	&&	$y = a \& b$	$y = a \&\& b$
OR			$y = a b$	$y = a b$
XOR	\wedge		$y = a \wedge b$	
NOT	\sim	!	$y = \sim a$	$y = !a$
upphöjt		**		$y = a^{**}b$

```

module and_gate(input a, input b, output y);
assign y = a & b;
endmodule

// Moduler - input/output
module modulnamn(input [1:0] in, output y)
// kod som beskriver kretsens beteende
endmodule

reg [1:0] stage; // 2 bitar bred, typen reg, variabelnamn state, bitvektor
state = {bit1, bit0}
state[1] = MSB
state[0] = LSB

// utan vektor
module not_32b(input in0, input in1, ... input in31, output out0, ... output
out31);
// kod som beskriver kretsens beteende
endmodule
// Med vektor
module not_32b(input [31:0] in, output [31:0] out);
endmodule

assign y = a & b; // kombinatoriska kretsar
övning
assign F = (A & B) | (!A & !B);
assign F = !(A^B); // xnor

```

```

// Instansiering - kombinatoriska kretsar
// xnor
module xnor_gate(input a, input b, output y);
    assign y = !(a ^ b);
endmodule
// xnor4
module four_xnor(input [3:0] a, input [3:0] b, output [3:0] y);
    xnor_gate xn0(a[0], b[0], y[0]);
    xnor_gate xn1(a[1], b[1], y[1]);
    xnor_gate xn2(a[2], b[2], y[2]);
    xnor_gate xn3(a[3], b[3], y[3]);
endmodule
// inbyggda primitiver
// and, or, xor, nand, nor, xnor

// Procedurer - AND-grind
// exempel
module and_gate(input a, input b, output reg y);

    // always, sensitivitet
    // gör detta när a eller b ändras
    // input behöver inte vara reg
    always @(a or b) begin
        // always @(*) begin gör detta när någon signal inom proceduren ändras

            if(a == 1'b1 & b == 1'b1) y = 1'b1;
            else y = 1'b0;
        end
    endmodule

    if(a == b) begin
        y = a;
        c = 1'b0;
    end else begin
        y = 1'b0;
        c = b;
    end

```

```

// Procedurer - case
module mux4(input [3:0] d, input [1:0] sel, output reg y);
    always @(d, sel) begin
        case(sel)
            2'b00: y = d[0];
            2'b01: y = d[1];
            2'b10: y = d[2];
            2'b11: y = d[3];
        endcase
    end
endmodule
module mux4(input [3:0] d, input [1:0] sel, output reg y);
    assign y = d[sel];
endmodule

// Procedurer - "Miniräknare"
module miniraknare(
    input [7:0] a, input [7:0] b,
    input [1:0] operation,
    output reg [7:0] y);
    always @(*) begin
        case(operation)
            2'b00: y = a + b; //plus
            2'b01: y = a - b; //Minus
            2'b10: y = a & b; //AND
            2'b11: y = a | b; //OR
        endcase
    end
endmodule

```

Siffror i verilog

Siffror - [bitar][bas][siffra]

<i>Bas</i>	<i>tecken</i>	<i>exempel</i>
2	'b	12'b010110100101
8	'o	12'o2645
10	'd	12'd1445
16	'h	12'h5A5

Kombinerade bitar

$$\{a, b, c[7:4], 1'b0\} \Rightarrow (\text{bits}) a \ b \ c7 \ c6 \ c5 \ c4 \ 0$$

```
// D-flip-flop
module simple_dff(input d, input clock, output reg q);
    always @(posedge clock) begin
        // posedge = rising edge    negedge = falling edge
        q <= d; // q tilldelas d just vid flank
    end
endmodule

// shiftregister
module shift_8b(input d, input clock, output [3:0] q);
    wire [3:0] next;
    assign q = next;
    simple_dff dff0( d, clock, next[0]);
    simple_dff dff1(next[0], clock, next[1]);
    simple_dff dff2(next[1], clock, next[2]);
    simple_dff dff3(next[2], clock, next[3]);
endmodule
```

```

// Interna signaler
module internal_signals(input a, output y);
    wire [7:0] connector;
    //Combinational logic

    reg [7:0] memory;
    // signal som behåller sitt värde genom tiden. lagringsvariabel
    // For sequential logic, används för interna tillstånd och minne
    // används i always block
endmodule

// skiftregister
module shift_8b(input d, input clock, output reg [7:0] q);
    always @(posedge clock) begin
        q <= {q[6:0],d};
    end
endmodule

// finite state machine
module simple_fsm(input up, input clock, output reg [1:0] state);
    always @(posedge clock) begin
        case(state)
            0: if(up) state <= 1;
                else state <= 3;
            1: if(up) state <= 2;
                else state <= 0;
            2: if(up) state <= 3;
                else state <= 1;
            3: if(up) state <= 0;
                else state <= 2;
        endcase
    end
endmodule

```

```

// finite state machine with reset, synkron
module simple_fsm(input up, input clock, input reset, output reg [1:0] state);
    parameter zero = 0, one = 1, two = 2, three = 3; // namngivna tillstånd
    always @(posedge clock) begin
        // always @(posedge clock, negedge reset) begin // asynkron reset
        // reset doesnt wait for edge

            if(!reset) state <= 0;    // reset aktiv låg
            // reset = 1 : kör fsm
            // reset = 0 : state = 0
            else begin
                case(state)
                    0: if(up) state <= one;
                        else state <= three;
                    1: if(up) state <= two;
                        else state <= zero;
                    2: if(up) state <= three;
                        else state <= one;
                    3: if(up) state <= zero;
                        else state <= two;
                endcase
            end
        end
    endmodule

```

```

// fsm - internt tillstånd
module simple_fsm(input up, input clock, input reset, output [7:0] lights);
    parameter zero = 0, one = 1, two = 2, three = 3;
    reg [1:0] state;
    always @(state) begin
        case(state)
            zero: lights = 8'b11111100;
            one: lights = 8'b11100011;
            two: lights = 8'b10001111;
            three: lights = 8'b00111110;
        endcase
    end
    always @(posedge clock) begin
        if(!reset) state <= 0;
        else begin
            case(state)
                0: if(up) state <= one;
                    else state <= three;
                1: if(up) state <= two;
                    else state <= zero;
                2: if(up) state <= three;
                    else state <= one;
                3: if(up) state <= zero;
                    else state <= two;
            endcase
        end
    end
endmodule

```

```

// räknare
module counter(input clock, input reset, input up_down, output [7:0] value);
    reg [7:0] count;
    assign value = count;
    always @(posedge clock) begin
        if(reset) count <= 0;
        else if(up_down == 1'b1) count = count + 1;
        else count <= count - 1;
    end
endmodule

```

```

// Designexempel - Garagedörr

module garagedoor(input reset,input clock, input open, input close,
                   input door_up, input door_down,
                   output reg power_up, output reg power_down);
parameter state_open = 0, state_closed = 1, state_opening = 2,
state_closing = 3;
reg [1:0] state;
always @(state) begin
  case(state)
    state_open: begin
      power_up = 1'b0;
      power_down = 1'b0;
    end
    state_closed: begin
      power_up = 1'b0;
      power_down = 1'b0;
    end
    state_opening: begin
      power_up = 1'b1;
      power_down = 1'b0;
    end
    state_closing: begin
      power_up = 1'b0;
      power_down = 1'b1;
    end
  endcase
end

always @(posedge clock) begin
  if(reset) begin
    state <= state_open;
  end else begin
    case(state)
      state_open: if(close) state <= state_closing;
      state_closed: if(open) state <= state_opening;
      state_opening: begin
        if(close) state <= state_closing;
        else if(door_up) state <= state_open;
      end
      state_closing: begin
        if(open) state <= state_opening;
        else if(door_down) state <= state_closed;
      end
    end
  end
end

```

```

        end
    endcase
end
end
endmodule

module door_simulator(input clock, input reset, input power_up,
                      input power_down, output reg open, output reg closed);
reg [6:0] position;
always @(position) begin
    case(position)
        0: begin
            open = 1'b0;
            closed = 1'b1;
        end
        100: begin
            closed = 1'b0;
            open = 1'b1;
        end
        default: begin
            open = 1'b0;
            closed = 1'b0;
        end
    endcase
end
always @(posedge clock) begin
    if(reset) position = 50;
    if(power_up && (position != 100)) begin
        position <= position + 1;
    end
    else if(power_down && (position != 0)) begin
        position <= position - 1;
    end
end
endmodule

module tb(input open, close, clock, reset, output power_up, power_down);
    wire opened, closed;
    garagedoor gd(reset, clock, open, close, opened, closed, power_up,
    power_down);
    door_simulator ds(clock, reset, power_up, power_down, opened, closed);
endmodule

```

