Statemaschine (Zustandsautomat)

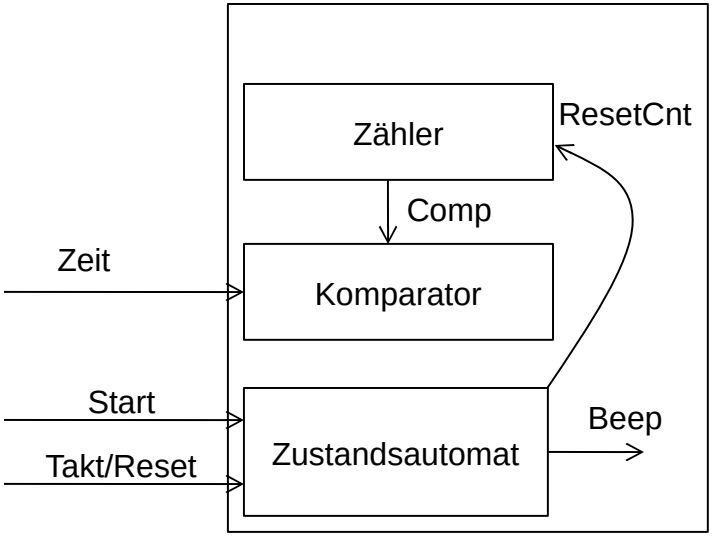
- Zustandsmaschinen werden für die Ansteuerung von digitalen Systemen verwendet. Man kann sie mit Programmen vergleichen - der Programmcode ist fest
- Ausgangssignale von Zustandsmaschinen (*und allen sequentiellen Schaltungen*) hängen nicht nur von momentanen Werten der Eingangsvariablen sondern auch von deren Reihenfolge. Dieses Verhalten ist möglich wenn die Schaltung Speicherelemente hat.
- Speicherelemente befinden sich in einem Zustand. Der nächste Zustand hängt vom momentanen Zustand und von den Eingangsvariablen.
- N Speicherelemente -> maximal 2^n Zustände
- Da es eine endliche Zahl von möglichen Zuständen gibt, nennt man solche Zustandsautomate finite-state Maschinen
- Den Zustand des Speicherelements nennt man Zustandsvariable.

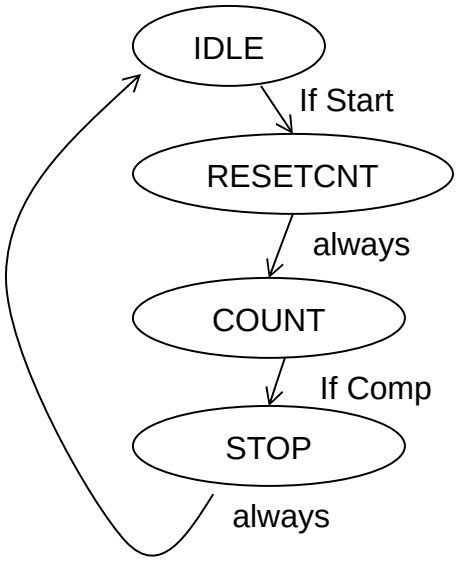
- Die Zustandsmaschinen kann man in zwei Klassen unterteilen.
- Moore Typ: Der Ausgang hängt nur von der Zustandsvariable – d.h. nur vom Zustand der Maschine
- Mealy Typ: Der Ausgang hängt auch von Eingängen
- Mealy Maschinen können oft mit weniger Zuständen realisiert werden, brauchen aber kombinatorische Schaltung für die Erzeugung von Ausgangssignalen. Moore Typ Automaten sind einfacher zu beschreiben, brauchen oft mehr Zuständen.

- Man kann in einem Zustandsautomat jede Art von Speicherzellen verwenden um den Zustand zu speichern
- Wenn alle Speicherzellen in einer Statemaschine den Zustand gleichzeitig ändern, z.B. auf steigende Taktflanke, nennen wir dieses Netzwerk synchron. Synchrone Zustandsmaschine verwenden Filp-Flops als Speicherelemente
- Man kann auch Zustandsmaschinen bauen, deren Zustand sich durch Änderung von verschiedenen Eingangssignalen ändert. Solche Netzwerke nennen wir asynchron. Asynchrone Zustandsmaschinen verwenden Latches als Speicherelemente.

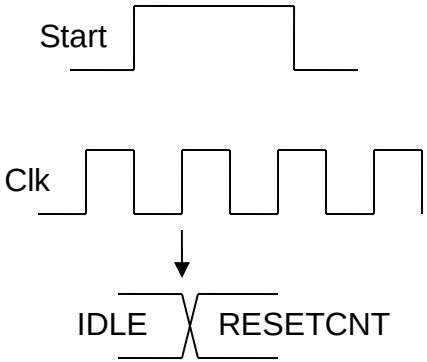
- Die Funktionalität einer Zustandsmaschine kann man z.B. mit einem Zustandsdiagramm beschrieben. Solch ein Zustandsdiagram hat für eine Statemaschine die gleiche Bedeutung wie eine Wahrheitstabelle für eine kombinatorische Schaltung.
- Ein Zustandsdiagramm kann entweder graphisch oder als Verilog/VHDL Code dargestellt werden.

- Beispiel Timer
- Der Timer besteht aus folgenden Komponenten – einem Zähler, einem Komparator, einem Startkopf, einem Drehregler für die Zeiteinstellung und einem Lautsprecher. Der Komparator vergleicht den Zähler-Zustand mit der eingestellten Zeit.
- Die Eingänge für die State-Maschine sind das Startsignal und der Komparator-Ausgang. Die Ausgangssignale sind ein Reset Signal für den Zähler und ein Signal für den Lautsprecher.
- Statmeschine braucht noch ein Taktsignal und asynchrons Reset

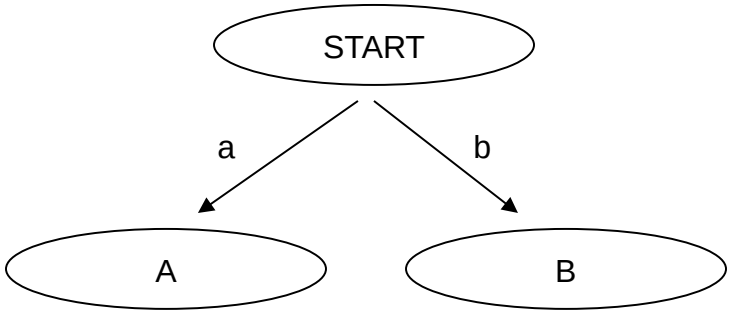




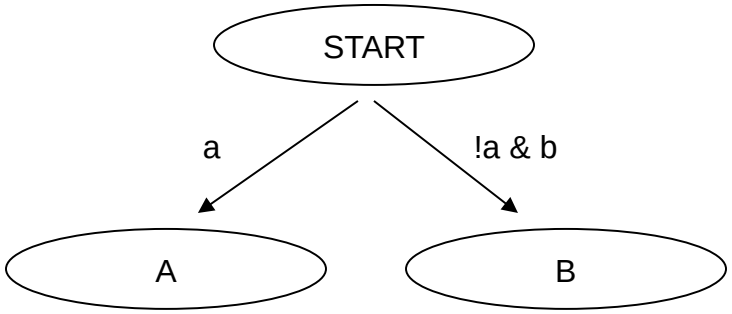
- Zustandsdiagramm
- Übergänge
- Bedingung für den Übergang
- Synchron -> Bedingung muss erfüllt werden, der Übergang passiert dann auf die nächste Taktflanke
- Annahme: Der Zustandsautomat bleibt in einem Zustand wenn die Bedingung für den Übergang nicht erfüllt ist
- Always: Übergänge die immer passieren



- Abzweigung
- Eindeutigkeit überprüfen



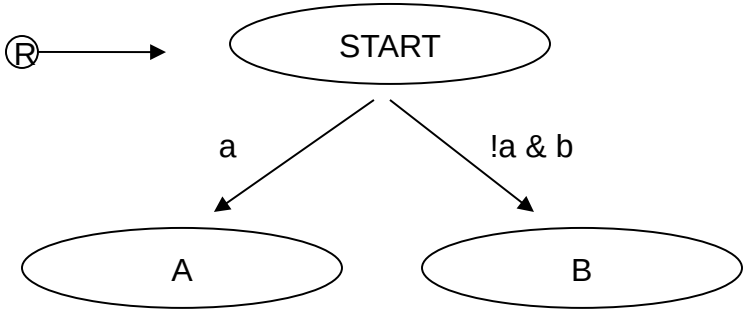
- Hierarchie der Eingänge definieren (a > b)
- Einfacher in HDL

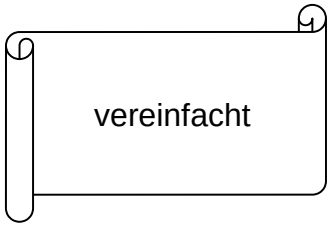


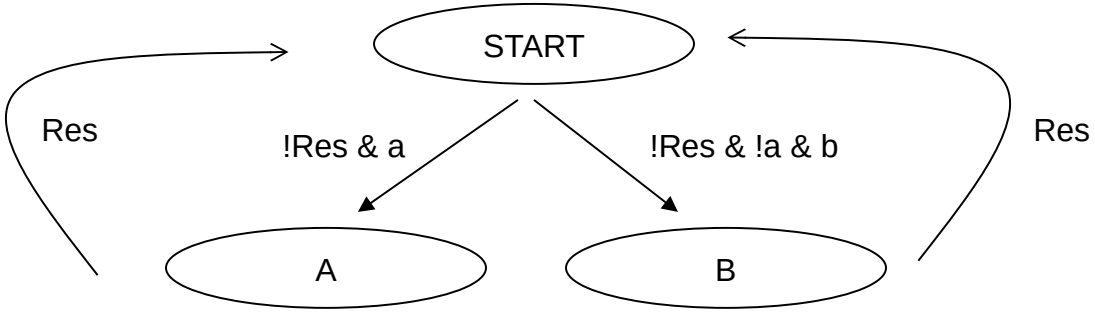

```
case  (State)
        START: begin
        if (a) State <= A;
        else if (b) State <= B;
        //else State <= START;
        A:
        B:
endcase
```

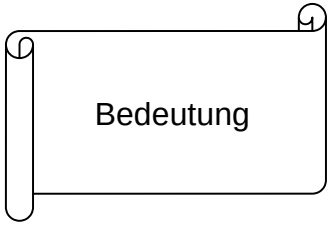
- „CASE“ Befehl
- Im Block „START“ werden alle Übergänge vom Zustand „START“ definiert.
- IF-ELSE Befehl definiert die Hierarchie
- Der letzte ELSE Block wäre ausgeführt wenn weder a noch b wahr sind
- Diese Zeile kann weggelassen werden, da Flipflops ihren Zustand behalten wenn es zu keiner Änderung kommt

- Reset-Eingang
- Wenn Reset = 1 ist, kommt der Automat in einen bestimmten Zustand
- Reset kann synchron oder synchron sein. Wenn es synchron ist, wird es wie ein normaler Eingang behandelt. Die Zustandsänderung passiert auf die Taktflanke. Wenn das Reset asynchron ist, passiert die Zustandsänderung auf Reset Flanke



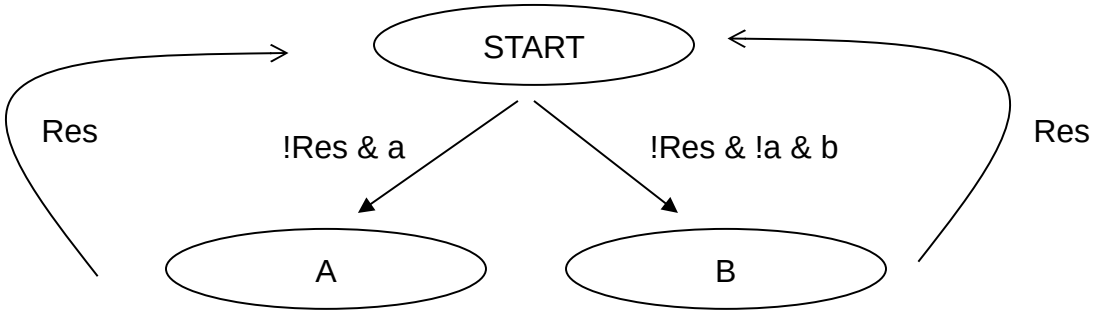


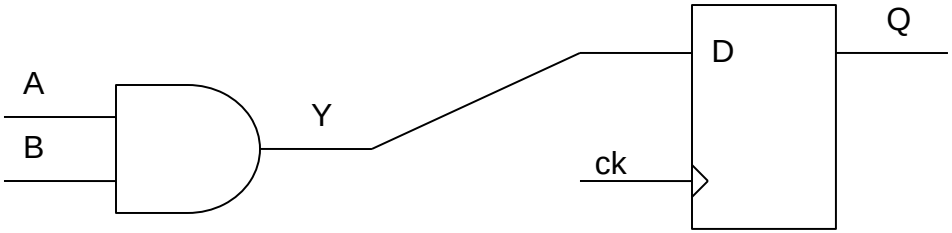




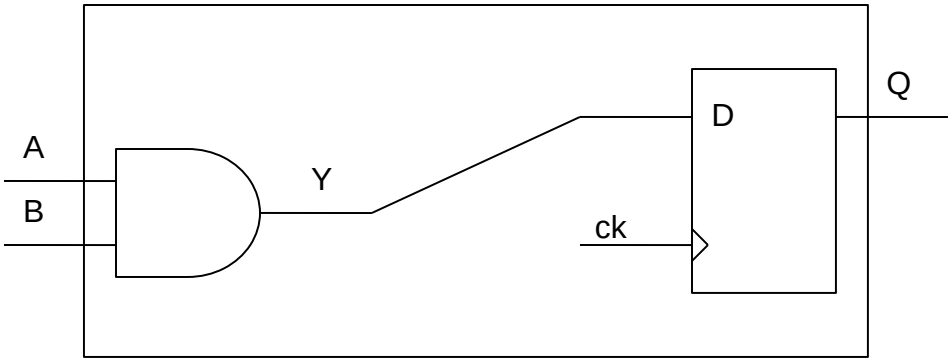
- Code ist übersichtlicher als der Zustandsdiagram. CASE Befehl wird nur bei Res = 0 ausgeführt.

```
if(Res) State <= START;
else begin
        case  (State)
                START: begin
                if (a) State <= A;
                else if (b) State <= B;
                //else State <= START;
                A:
                B:
                …
        endcase
end//not reset
```

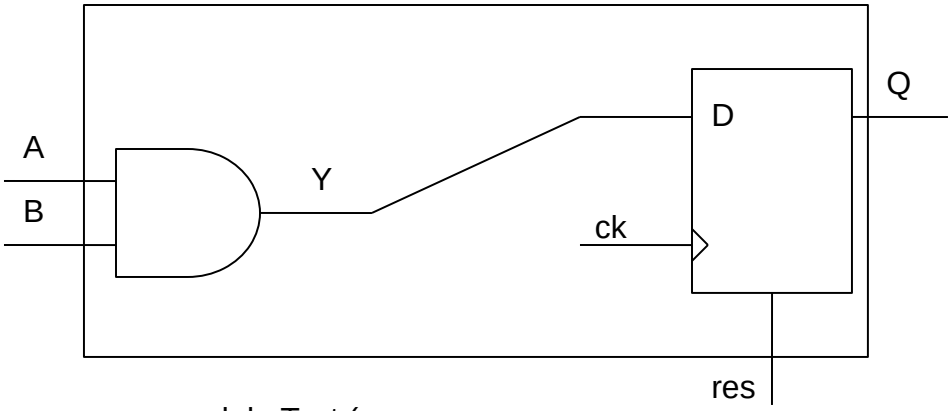





```
reg Q;
wire A, B, Y;

assign Y = A & B;

always @ (posedge clk) begin

        Q <= Y;
        //Q <= A & B;

        end
```




```
module Test (
   input A,
   input B,
   output reg Q
);


always @ (posedge clk) begin

        Q <= A & B;

        end
        endmodule
```




```
module Test (
   input A,
   input B,
   input res,
   output reg Q
);
always @ (posedge clk or posedge res) begin
        if (res) Q <= 0;
        else begin
                Q <= A & B
        end//res
        end//always
        endmodule
```

```
module Timer (
  input clk,
  input reset,
  input start,
  input comp,
  output wire resetcounter,
  output wire beep
);
reg [1:0] State;

parameter IDLE = 2'b00, RESETCNT = 2'b01, COUNT = 2'b11, STOP = 2'b10;

assign resetcounter = {State == RESETCNT};
assign beep = {State == STOP};

always @ (posedge clk or posedge reset) Begin
if (reset) State <= IDLE;
else begin
        case (State)
                IDLE: begin
                        if (Start) State <= RESETCNT;
                        //!Else State <= IDLE;
                end
                RESETCNT: begin
                        State <= COUNT;
                        //Counter <= 0;
                end
                COUNT: begin
                        //Counter <= Counter + 1;
                        if (comp) State <= STOP;
                end
                STOP: begin
                        State <= IDLE;
                end
        endcase
end//not reset
end//always

endmodule
```

Module Definition, Eingänge, Ausgänge

Typen von Hardware-Komponenten: reg, wire, logic (system verilog)

Kein Komma

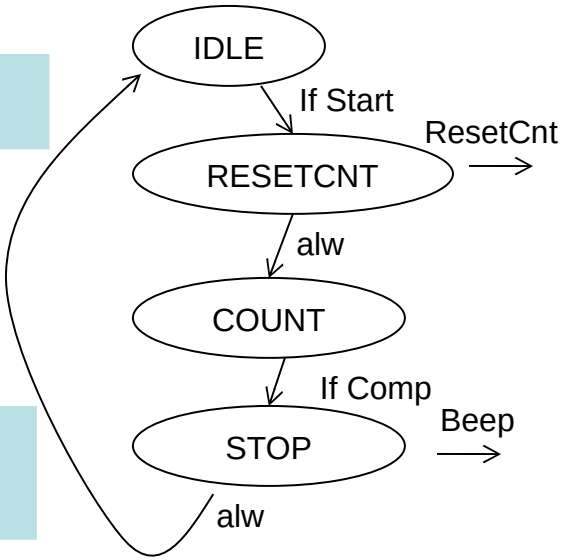
Parameter-Definition

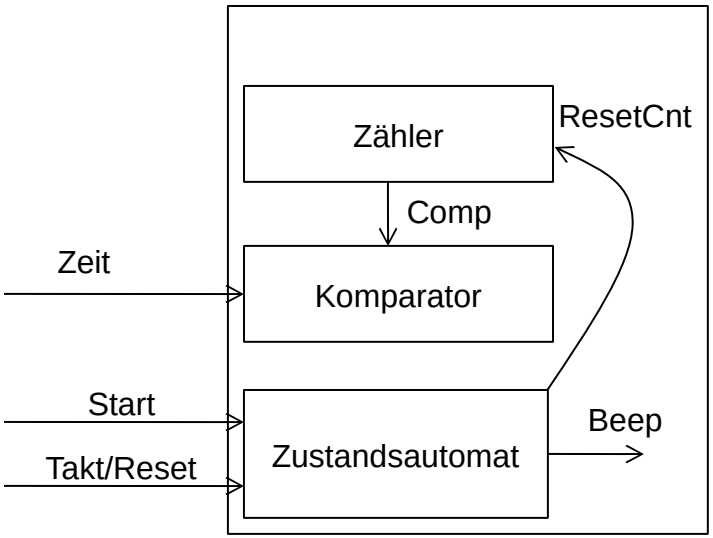
„Always“ Block

Asynchrones reset

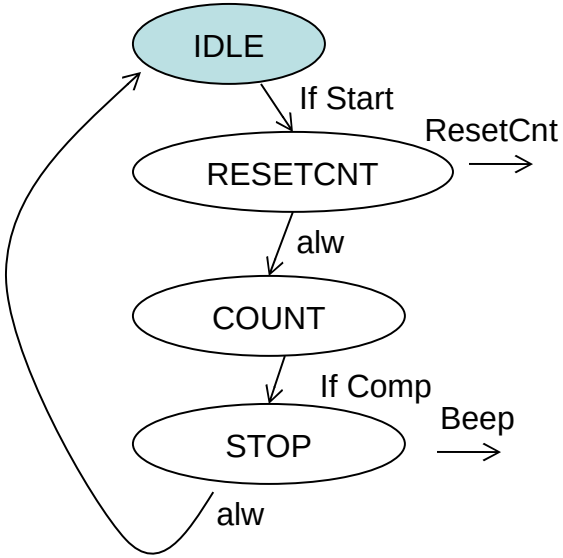
“Assign” Befehl (Kombinatorische Logik)

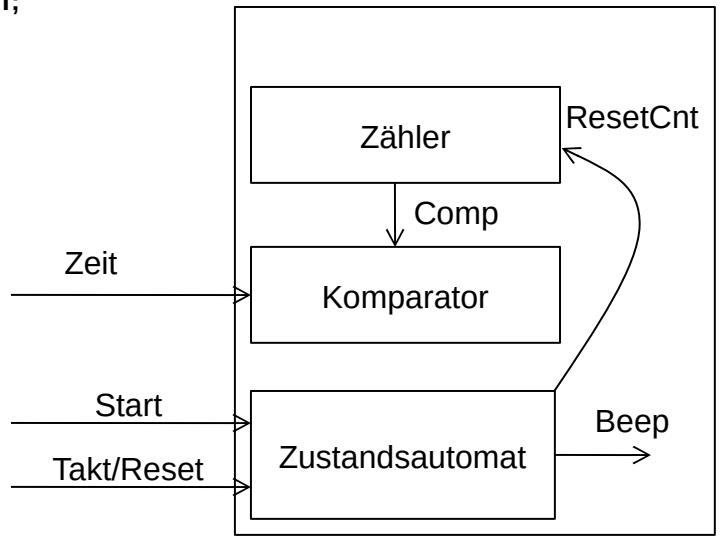
nonblocking assignment “<=“



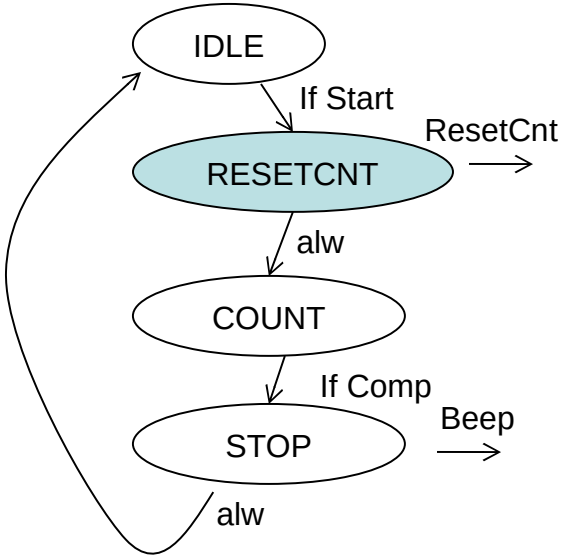


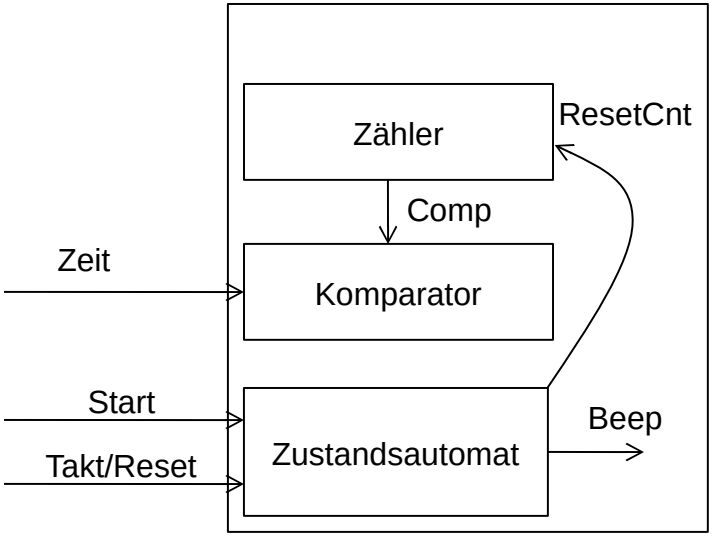
```
module Timer (
  input clk,
  input reset,
  input start,
  input comp,
  output wire resetcounter,
  output wire beep
);
reg [1:0] State;

parameter IDLE = 2'b00, RESETCNT = 2'b01, COUNT = 2'b11, STOP = 2'b10;

assign resetcounter = {State == RESETCNT};
assign beep = {State == STOP};

always @ (posedge clk or posedge reset) Begin
if (reset) State <= IDLE;
else begin
        case (State)
                IDLE: begin
                        if (Start) State <= RESETCNT;
                        //!Else State <= IDLE;
                end
                RESETCNT: begin
                        State <= COUNT;
                        //Counter <= 0;
                end
                COUNT: begin
                        //Counter <= Counter + 1;
                        if (comp) State <= STOP;
                end
                STOP: begin
                        State <= IDLE;
                end
        endcase
end//not reset
end//always

endmodule
```



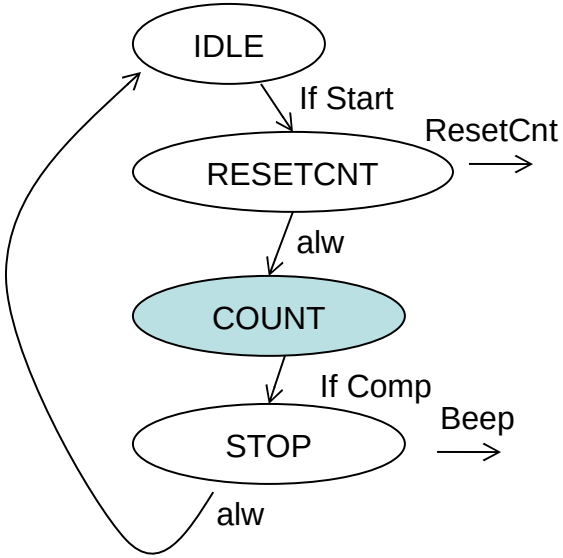


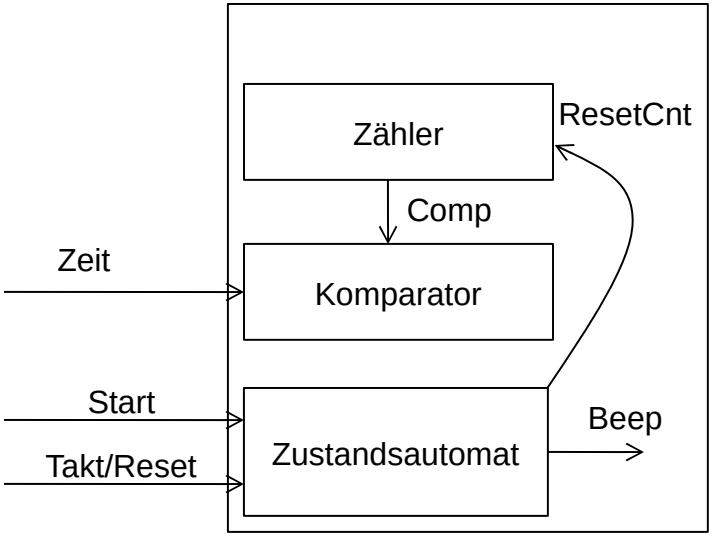
```
module Timer (
  input clk,
  input reset,
  input start,
  input comp,
  output wire resetcounter,
  output wire beep
);
reg [1:0] State;

parameter IDLE = 2'b00, RESETCNT = 2'b01, COUNT = 2'b11, STOP = 2'b10;

assign resetcounter = {State == RESETCNT};
assign beep = {State == STOP};

always @ (posedge clk or posedge reset) Begin
if (reset) State <= IDLE;
else begin
        case (State)
                IDLE: begin
                        if (Start) State <= RESETCNT;
                        //!Else State <= IDLE;
                end
                RESETCNT: begin
                        State <= COUNT;
                        //Counter <= 0;
                end
                COUNT: begin
                        //Counter <= Counter + 1;
                        if (comp) State <= STOP;
                end
                STOP: begin
                        State <= IDLE;
                end
        endcase
end//not reset
end//always

endmodule
```



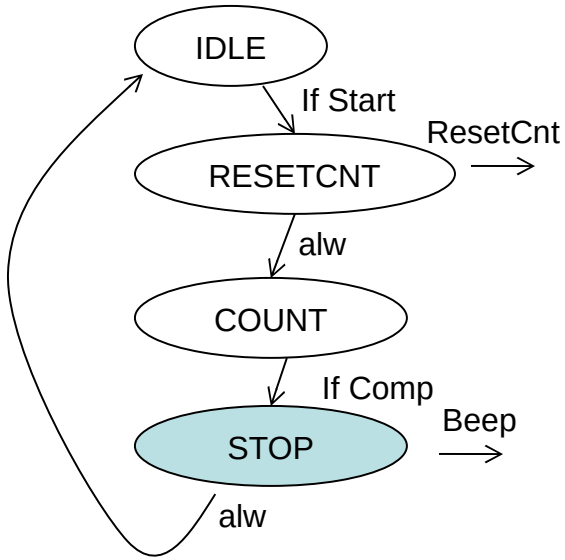


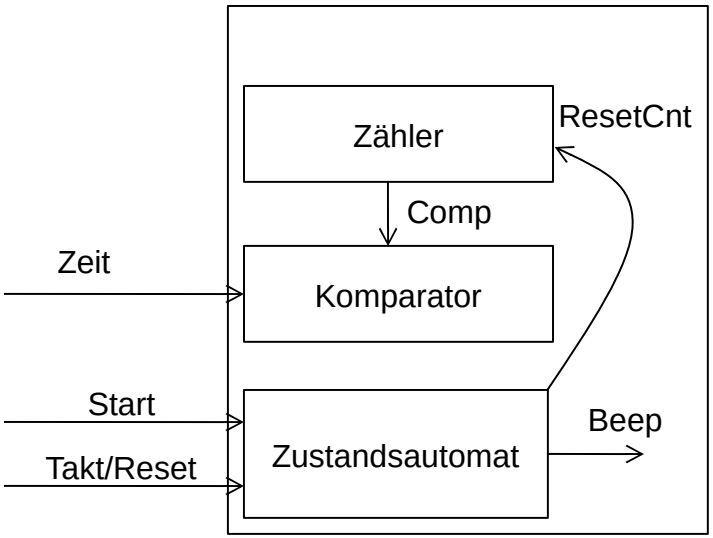
```
module Timer (
  input clk,
  input reset,
  input start,
  input comp,
  output wire resetcounter,
  output wire beep
);
reg [1:0] State;

parameter IDLE = 2'b00, RESETCNT = 2'b01, COUNT = 2'b11, STOP = 2'b10;

assign resetcounter = {State == RESETCNT};
assign beep = {State == STOP};

always @ (posedge clk or posedge reset) Begin
if (reset) State <= IDLE;
else begin
        case (State)
                IDLE: begin
                        if (Start) State <= RESETCNT;
                        //!Else State <= IDLE;
                end
                RESETCNT: begin
                        State <= COUNT;
                        //Counter <= 0;
                end
                COUNT: begin
                        //Counter <= Counter + 1;
                        if (comp) State <= STOP;
                end
                STOP: begin
                        State <= IDLE;
                end
        endcase
end//not reset
end//always

endmodule
```





```
module Timer (
  input clk,
  input reset,
  input start,
  input comp,
  output wire resetcounter,
  output wire beep
);
reg [1:0] State;

parameter IDLE = 2'b00, RESETCNT = 2'b01, COUNT = 2'b11, STOP = 2'b10;

assign resetcounter = {State == RESETCNT};
assign beep = {State == STOP};

always @ (posedge clk or posedge reset) Begin
if (reset) State <= IDLE;
else begin
        case (State)
                IDLE: begin
                        if (Start) State <= RESETCNT;
                        //!Else State <= IDLE;
                end
                RESETCNT: begin
                        State <= COUNT;
                        //Counter <= 0;
                end
                COUNT: begin
                        //Counter <= Counter + 1;
                        if (comp) State <= STOP;
                end
                STOP: begin
                        State <= IDLE;
                end
        endcase
end//not reset
end//always

endmodule
```



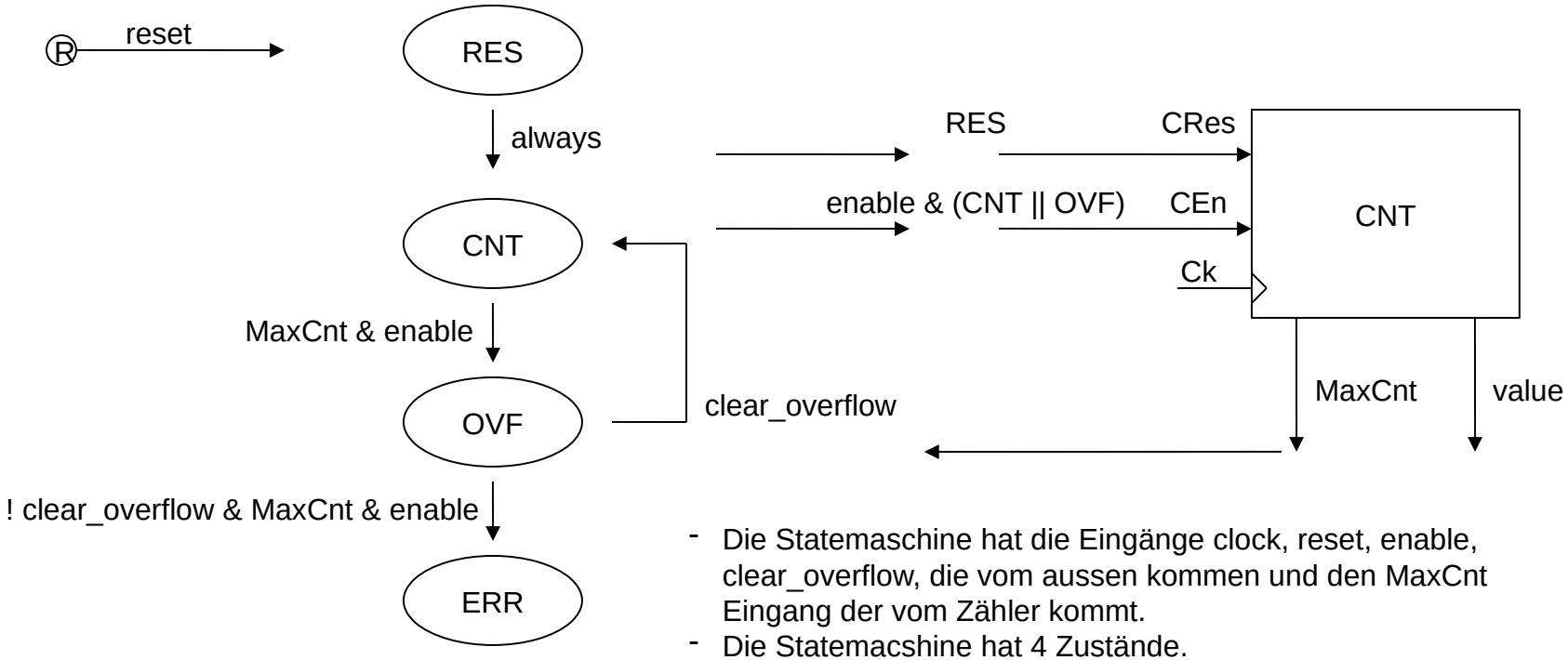


- Oft enthält der Code der Statemaschine auch die Digitalschaltungen, die die Statemaschine ansteuert.
- Gray Code verwendet

- Übung
- Es soll ein Zäher implementiert werden der zählt wenn Signal enable = 1 ist.
- Wenn es zum ersten Überlauf kommt (Übergang MaxCnt->0) wird Signal overflow auf 1 gesetzt.
- Wenn overflow = 1 ist, und es kommt zum zweiten Überlauf wird overflow_error = 1.
- overflow wird mit dem Eingang clear_overflow zurückgesetzt.
- Es gibt auch ein reset Eingang der sowohl overflow, overflow_error als auch den Zähler zurücksetzt.
- Der Zähler zählt nicht wenn overflow_error = 1 und wenn reset aktiv ist.

Ck, reset, enable, clear_overflow

overflow, overflow_error, value



- Die Statemaschine hat die Eingänge clock, reset, enable, clear_overflow, die vom aussen kommen und den MaxCnt Eingang der vom Zähler kommt.
- Die Statemacshine hat 4 Zustände.
- Ausgänge overflow und overflow_error werden erzeugt wenn die Maschine im OVF/ERR Zustand ist.
- Die Statemaschine erzeugt auch die CEn und CRes Signale für den Zähler.
- Beachten wir, dass En sowohl von Zuständen als auch vom Eingang abgängig ist. Es handelt sich hier (in Bezug auf diese Ausgänge) um eine Mealy – Zustandsmaschine.

```
module Counter #(parameter WIDTH = 8)(
  input clock,
  input reset,
  input enable,
  // Note the "reg" definition to value
  output reg [WIDTH - 1:0] value,
  output wire overflow,
  input clear_overflow,
  input reinit,
  output wire overflow_error,
);
reg [1:0] State;
wire MaxCnt;
parameter RES = 2'b00, CNT = 2'b01, OVF = 2'b11, ERR = 2'b10;
assign overflow = {State == OVF};
assign overflow_error = {State == ERR};
assign MaxCnt = {value == {WIDTH {1'b1}}};

always @ (posedge clock) begin
        if (reset | reinit) begin
                State <= RES;
                value <= 0;
        end
        else begin
                case (State)
                        RES: begin
                                State <= CNT;
                        end
                        CNT: begin
                                if(enable) value <= value + 1;//counter
                                if(MaxCnt & enable) State <= OVF;
                        end
                        OVF: begin
                                if(enable) value <= value + 1;//counter
                                if(clear_overflow) State <= CNT;
                                else if(MaxCnt & enable) State <= ERR;
                        end
                        ERR: begin
                                State <= ERR;
                        end
                endcase
        end//not reset
end//always
endmodule
```

Zähler-Länge als Parameter

Module Definition, Eingänge, Ausgänge

Parameter-Definition

Ausgänge als Funktionen von Zusänden

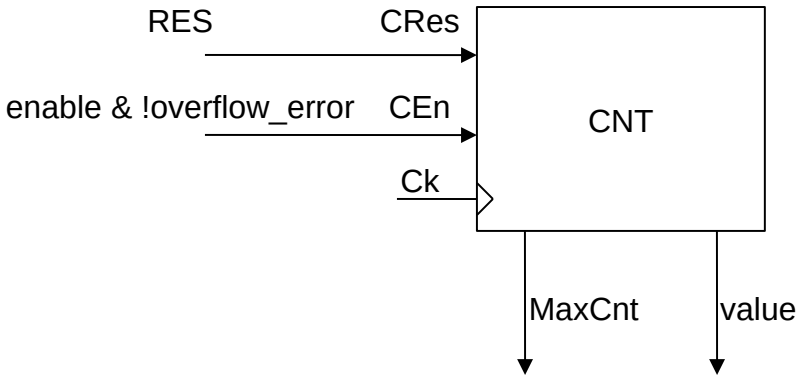
„Always“ Block, synchrones reset

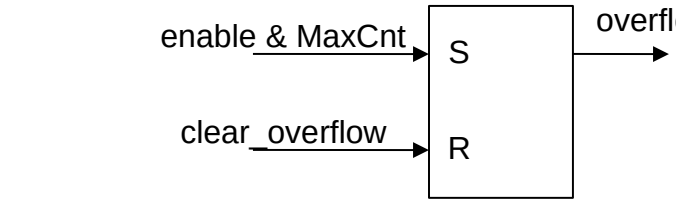
```
module Counter #(parameter WIDTH = 8)(
  input clock,
  input reset,
  input enable,
  // Note the "reg" definition to value
  output reg [WIDTH - 1:0] value,
  output wire overflow,
  input clear_overflow,
  input reinit,
  output wire overflow_error,
);
enum reg [1:0] {RES = 2'b00, CNT = 2'b01, OVF = 2'b11, ERR = 2'b10} State;
wire MaxCnt;
assign overflow = {State == OVF};
assign overflow_error = {State == ERR};
assign MaxCnt = {value == {WIDTH {1'b1}}};

always @ (posedge clock) begin
                if (reset | reinit) begin
                                State <= RES;
                                value <= 0;
                end
                else begin
                                case (State)
                                                RES: begin
                                                                State <= CNT;
                                                end
                                                CNT: begin
                                                                if(enable) value <= value + 1;//counter
                                                                if(MaxCnt & enable) State <= OVF;
                                                end
                                                OVF: begin
                                                                if(enable) value <= value + 1;//counter
                                                                if(clear_overflow) State <= CNT;
                                                                else if(MaxCnt & enable) State <= ERR;
                                                end
                                                ERR: begin
                                                                State <= ERR;
                                                end
                                endcase
                end//not reset
end//always
endmodule
```

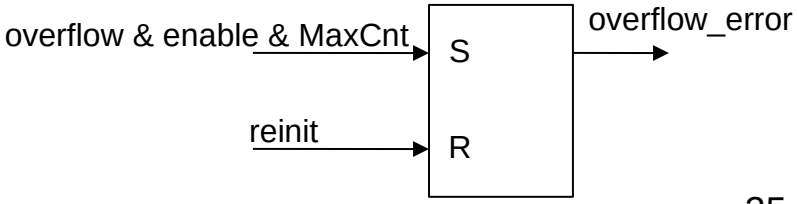
Trick: system verilog erlaubt uns “State” als eine reg - “Enumeration” zu deklarieren. So kann der Simulator die Namen der Zustände zeigen, was übersichtlicher ist.

```
module Counter #(parameter WIDTH = 8)(

  input clock,
  input reset,
  input enable,
  // Note the "reg" definition to value
  output reg [WIDTH - 1:0] value,
  output reg overflow,
  input clear_overflow,
  input reinit,
  output reg overflow_error,
  output reg [WIDTH - 1:0] value_sr


);


always @(posedge clock) begin

  if (reset) begin
                        value <= 0;
                        overflow <= 0;
                        overflow_error <= 0;
                        value_sr <= 0;
  end
  else if (reinit) begin
                        value <= 0;
                        overflow <= 0;
                        overflow_error <= 0;
                        value_sr <= 0;
  end
  else begin

    if(enable) if (!overflow_error) value <= value + 1;

    if (clear_overflow == 1) overflow <= 0;
    else if ( {value == {WIDTH {1'b1}}} && {enable == 1} ) overflow <= 1;


    if (overflow == 1) if ( {value == {WIDTH {1'b1}}} && {enable == 1} ) overflow_error <= 1;

    if(enable) value_sr <= {value_sr[WIDTH-2:0],~value_sr[WIDTH-1]};

  end

end

endmodule
```

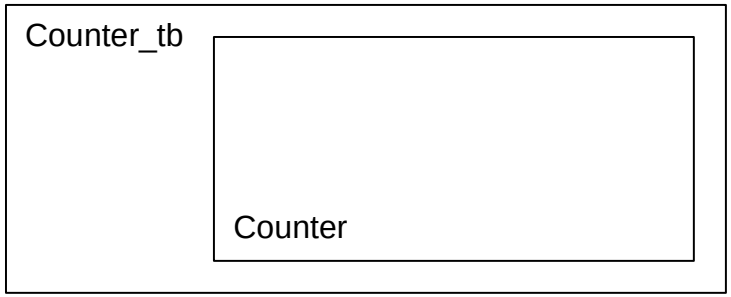
Weitere Zähler-Ralisierung mit zwei „Flags“



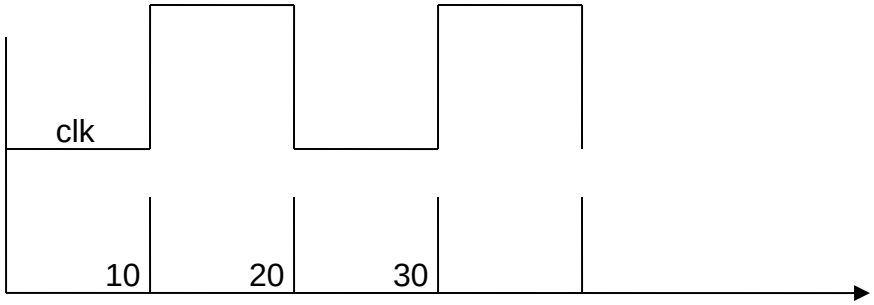




- Test bench ist ein Top-Module das verwendet wird um den Zähler zu simulieren
- Es soll alle Eingangs-Signale für den Zähler erzeugen
- Dabei sollen alle Eigenschaften und Funktionen vom Zähler simuliert werden:
- Ob sich der Zähler reseten lässt (reset - Funktionalität)
- Ob der Zähler zählt, wenn clock und enable aktiv sind (clock, enable - Funktionalität)
- Ob der Zähler in overflow-Zustand kommt, ob sich overflow reseten lässt (overflow, clear_overflow)
- Ob der Zähler in overlflow_error-Zustand kommt, ob sich overflow_error reseten lässt

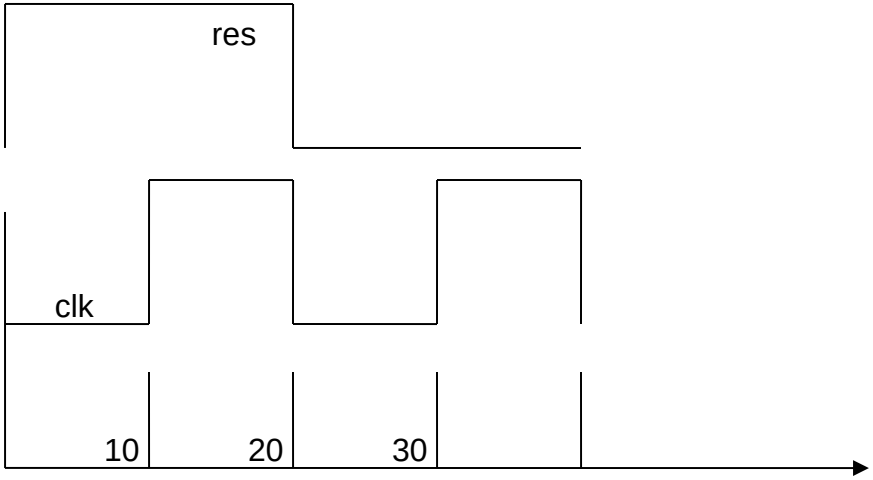


- Die Signale werden mit “blocking assignments” innerhalb von “initial” und “always” Blöcken generiert
- Takt-Erzeugung



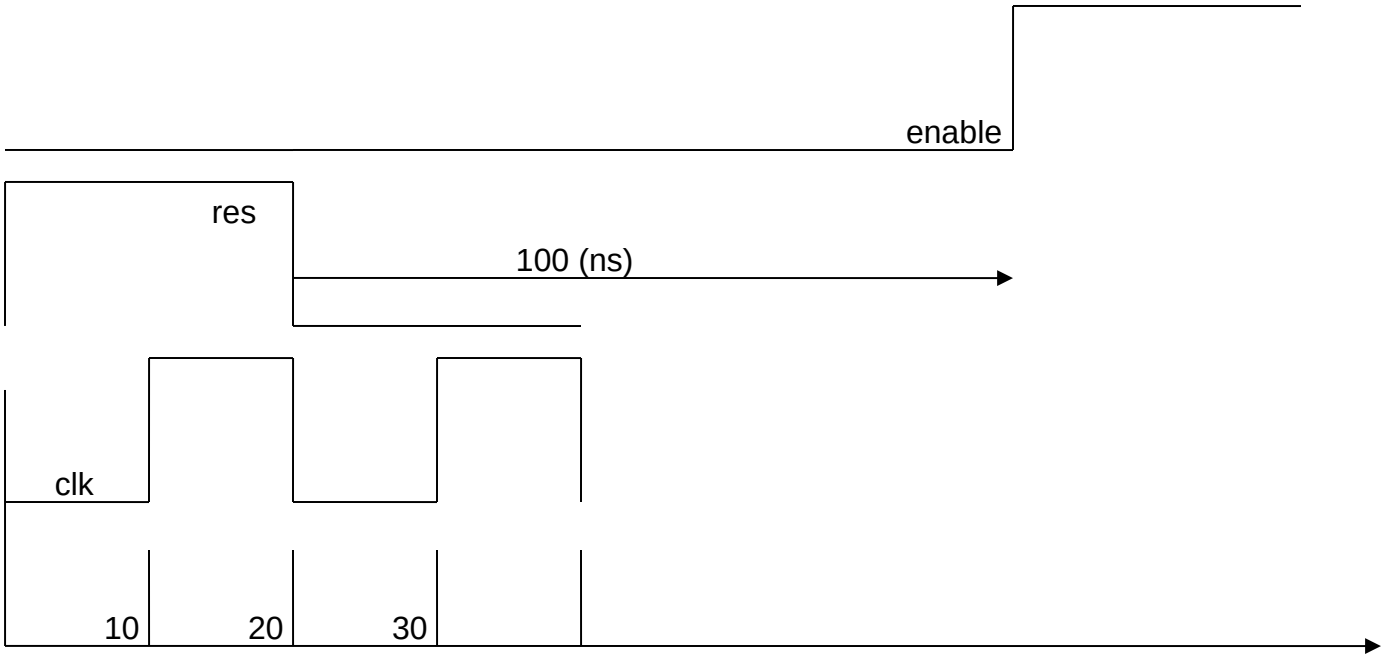

```
initial begin
                clk = 0; // 0 Value
end
always begin
                #10 clk <= ~ clk;
end
```

- Reset-Erzeugung
- Ein synchrones reset-Signal muss die steigende Taktflanke enthalten
- “Wait” Befehlt sichert die entsprechende Reset-Länge:
- Es wird gewartet bis res = 1 (A), dann auf posedge (B) und dann negedge clk-Flanke (B)
- Erst dann wird reset = 0 (C)



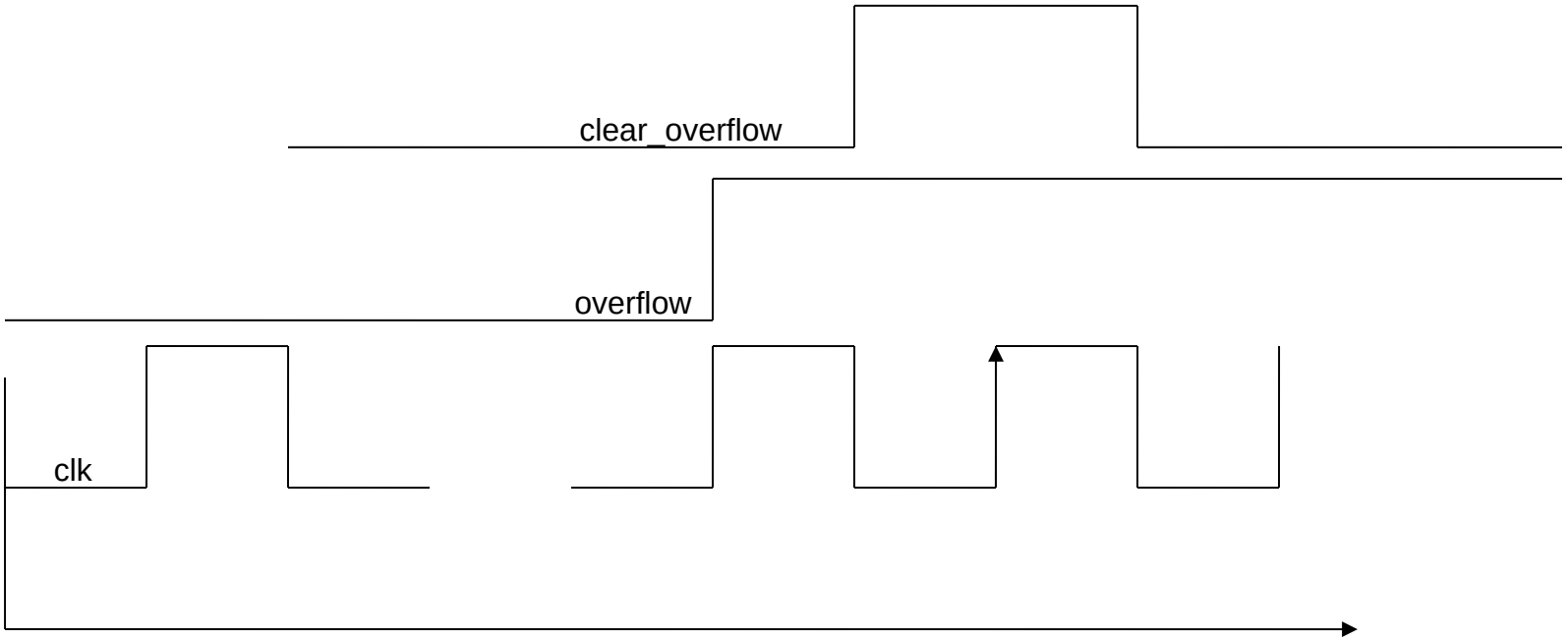

```
initial begin
                    res = 1;
wait (res==1);//A
@(posedge clk);//B
@(negedge clk);//C
res = 0;//D
end
```

- #100 (A) definiert die Wartezeit zwischen res=0 und enable=1




```
initial begin
                    res = 1;
wait (res==1);
@(posedge clk);
@(negedge clk);
res = 0;
#100 enable = 1;//A
```

- Clear_overflow wird auf erste fallende clk-Flanke nach overflow=1 generiert, es dauert bis zur nächsten fallenden clk-Flanke




```
initial begin
        wait (overflow==1);
        @(negedge clk);
        clear_overflow = 1;
        @(posedge clk);
        @(negedge clk);
        clear_overflow = 0;
        end
```