Entwurf Digitaler Systeme für FPGAs Praktikum

Verilog Crash-Kurs

Basierend auf dem Verilog Tutorial der Princeton University

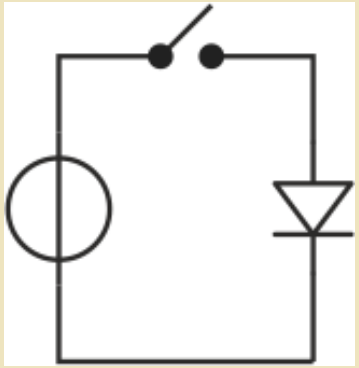
Was ist ein Zustandsautomat?

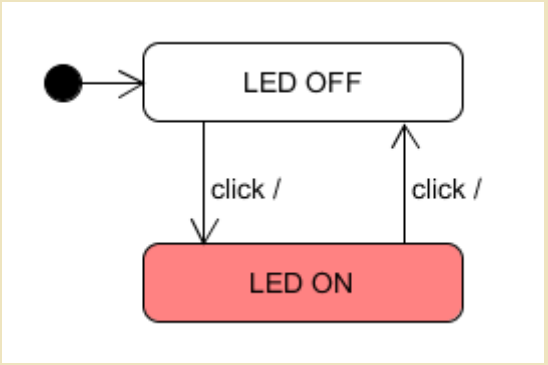
So gut wie jedes eindeutige System, jeder Prozess und jede Maschine lässt sich, bezogen auf sein Verhalten, durch einzelne **Zustände** beschreiben. Eine Ansammlung von Zuständen, welche zueinander in Beziehung stehen, bilden einen **Zustandsautomaten** (state machine).

Ist die Anzahl an möglichen Zuständen geschränkt, so spricht man von einem „Endlichen Zustandsautomaten“ (finit state machine => **FSM**)

Systembeschreibung:
LED mit Schalter

Vorgehen:
1. States extrahieren:
   <AN, AUS>
2. Transitionsbedingungen angeben:
   <click>
3. Ausgabewerte erstellen:
   >LED_ON, LED_OFF>





Modellierung von Zustandsautomaten

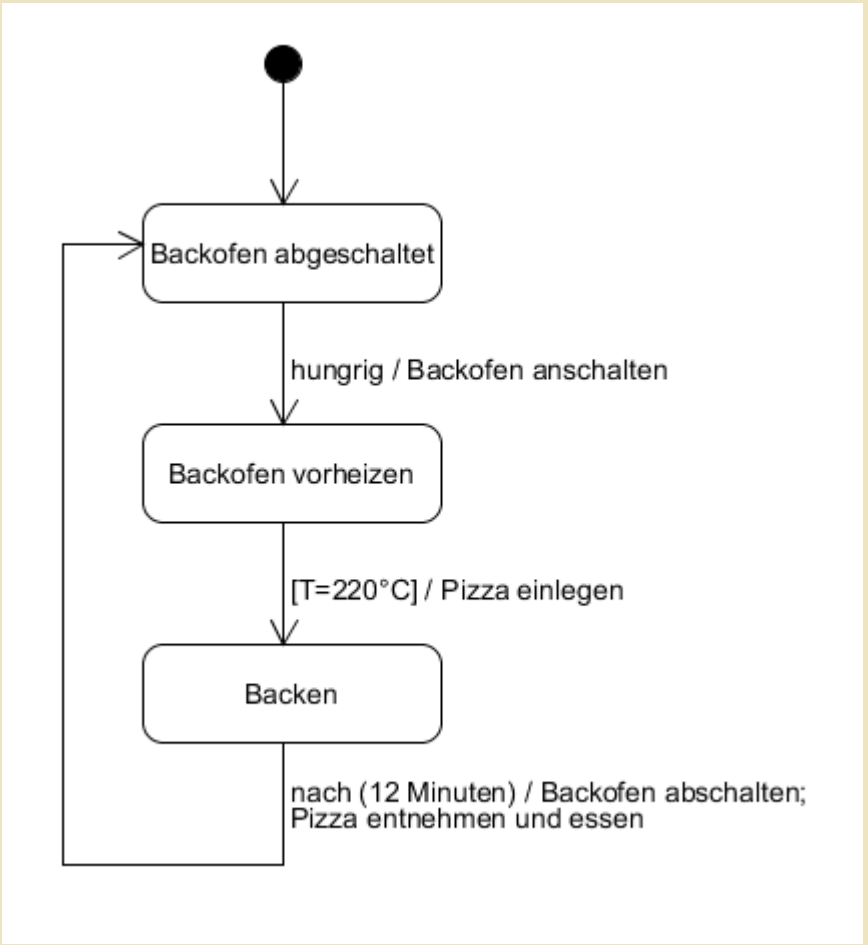
Beispiel: Tiefkühlpizza

Systembeschreibung:
Zubereitung einer TK-Pizza:

- Ober-Unterhitze: auf 220 °C vorheizen
- Backzeit: ca. 12 min

Vorgehen:

1. States extrahieren:
   <Backofen aus, Vorheizen, Backen>
2. Transitionsbedingungen angeben:
   <hungrig, T=220 °C, 12 min>
3. Ausgabewerte erstellen:
   <…>

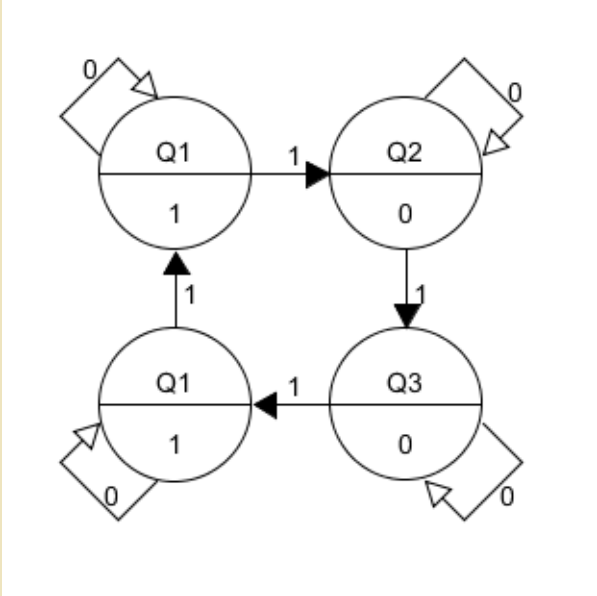


Modellierung von Zustandsautomaten

Beispiel: Sequenzgenerator (Moore Automat)

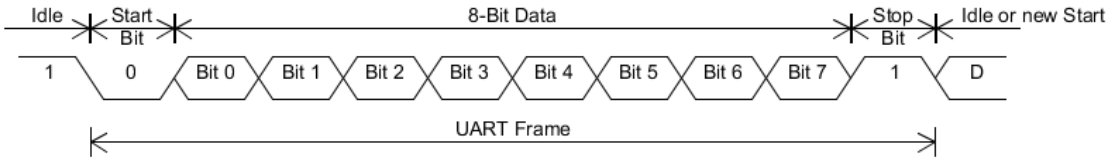
Systembeschreibung:

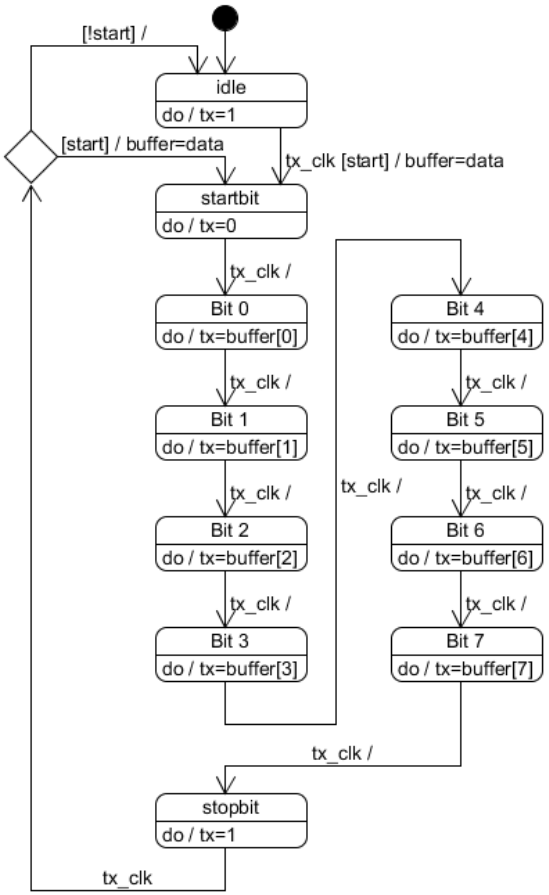
- Wiederkehrendes Muster von 100110011001…
- Eingang steuert die Transitionen
- State definiert den Ausgang



Modellierung von Zustandsautomaten

Beispiel: UART-Transmission





Systembeschreibung:

- eine TX-Leitung
- serieller Transfer von 8-Datenbits
- Start- und Stop-Bits

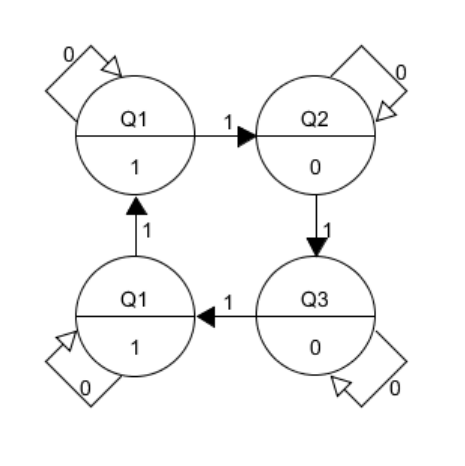
Vorgehen (eine Variante):

- jedes Bit auf der Leitung bekommt einen State
- Transition bei jedem Taktwechsel
- Zwischenspeicherung der Daten (Buffer)

Notationsformen (Historische Vertreter)

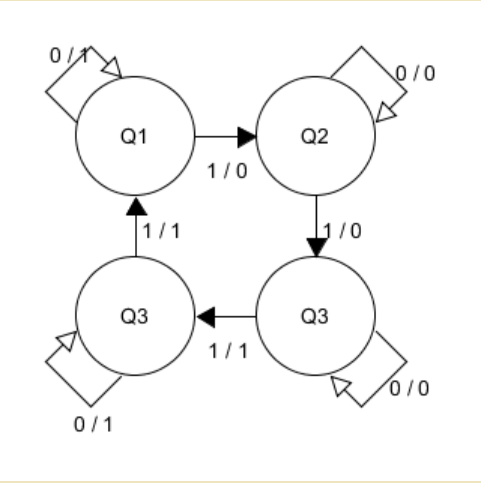
Moore-Automaten:
- Runde States mit Querstrich
- Eingangswerte stehen an der Transition
- Ausgangswerte stehen im State

Charakteristik:
=> Ausgang ist nur vom State abhängig



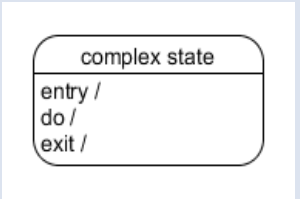
Mealy-Automaten:
- Runde States
- Eingangs- und Ausgangswerte stehen an der Transition
- Ausgang ist abhängig

Charakteristik:
=> Ausgang vom Eingang und dem State abhängig

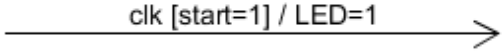


UML State-Machine Notation (Auszug)

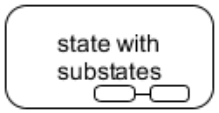
State / Complex State:
entry / <action> => beim Betreten des States
do / <action> => solange der State aktiv ist
exit / <action> => beim Verlassen des States



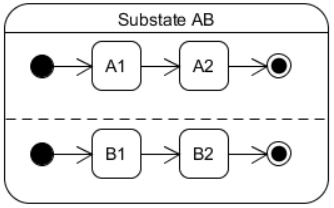
Transition:
<triggers> [<guard>] / <action>



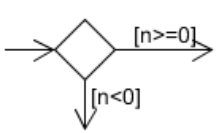
Superstate:
Oberinstanz



Substate:
mit 2 „Regionen“
(orthogonale Ausführung)



Entscheidungen:
mit „guard“-Notation



Start und Ende

Abbildung der FSM auf ein digitales System

Beispiel:
Es soll ein taktsynchroner Aufwärts-Ringzähler erstellt werden.
Das System hat zwei unterschiedliche Zählerbereiche. Die Auswahl der Bereiche erfolgt über einen Eingang „in“. Dabei steht der Wert „in“=1 für den Zählbereich von 0 bis 1 und „in“=0 für 0 bis 3. Eine Änderung des Zählbereichs darf dabei keinen Sprung in der Zählung bewirken. Die Ausgabe der Zahl soll über die Bitposition erfolgen (0=>0001;1=>0010…)

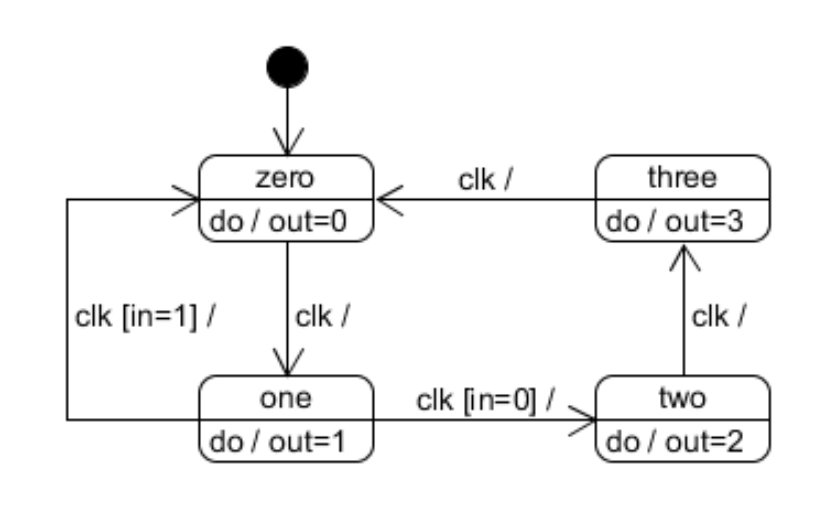
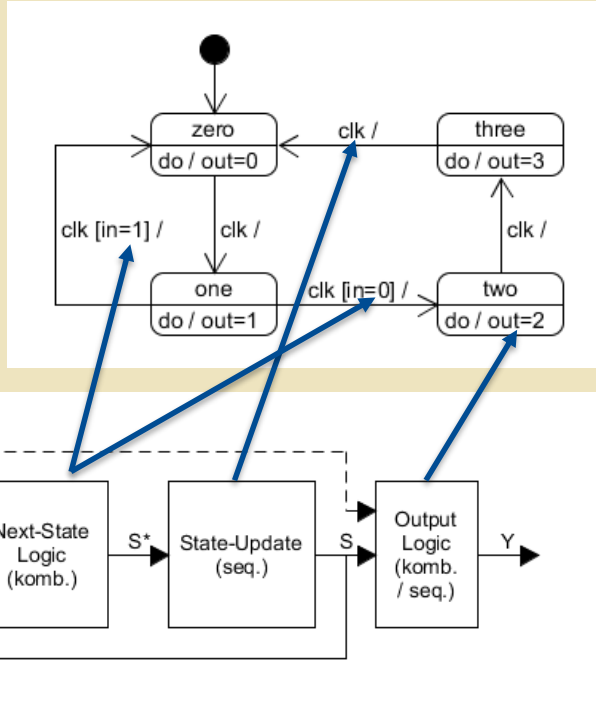


Abbildung der FSM auf ein digitales System (1)



Erste Näherung über die Automatenbeschreibung

- Moore-Automaten: S* = f(X,S), Y = f(S)
  (+) nur definierte Zustände am Ausgang (keine Rippel)
  (-) eventuell langsamer durch Zwischenspeicherung
- Mealy-Automaten: S* = f(X,S), Y = f(X,S)
  (+) dyn. Reaktion möglich, Zwischenberechnungen
  (-) Rippel können Systemverhalten stören

mit S:=State, S*:=Next-State, X:=Eingang, Y:=Ausgang

=> System ist ein **Moore**-Automat

Es entstehen **3** Hauptbereiche:

- Next-State Logic => Berechnung des nächsten gültigen Zustands (*kombinatorisch*)
- State-Update => Übernahme des Nachfolge-Zustands zum Zeitpunkt „t“ (*sequentiell*)
- Output-Logic => Individuelles Ausgangsmuster je nach Zustand (moore)
  bzw. Zustand und Eingangswert (mealy) (*kombinatorisch / sequentiell*)

Abbildung der FSM auf ein digitales System (2)

```
module fsm_moore(   input    clk,in,reset_n,
                    output reg [3:0] out);
reg [1:0] next_state;
reg [1:0] state;
parameter zero=0, one=1, two=2, three=3;

// Next-State Logic
always @(*)
begin
        //code
end

// State-Update Logic
always @(posedge clk or negedge reset_n)
begin
        //code
end

// Output-Logic
always @(*)
begin
        //code
end
endmodule
```

Grundstruktur:

- 3-teiliger Aufbau
- Next-State und Output sind kombinatorische Logik
- State-Update ist sequentielle Logik
- Definition der 4 möglichen Zustände: <zero,one,two,three>

Abbildung der FSM auf ein digitales System (3)

```
module fsm_moore(   input    clk,in,reset_n,
                    output reg [3:0] out);
reg [1:0] next_state;
reg [1:0] state;
parameter zero=0, one=1, two=2, three=3;

// Next-State Logic
always @(*)
begin
        //code
end

// State-Update Logic
always @(posedge clk or negedge reset_n)
begin
        //code
end

// Output-Logic
always @(*)
begin
        //code
end
endmodule
```

```
// Next-State Logic
always @(*)
begin
        //code
        case(state)
        zero:
                next_state<=one;
        one:
                if(in)
                   next_state<=zero;
                else
                   next_state<=two;
        two:
                next_state<=three;
        three:
                next_state<=zero;
        default:
                next_state<=zero;
        endcase

end
```

Transitionsbedingungen:
Wenn in State „one“ und „in“=1 gehe zurück zu State „zero“, sonst weiter zu State „two“.

Hinweis:
Durch die kombinatorische Ausführung wird der „next_state“ permanent aktualisiert. Die Zuweisung erfolgt aber erst später.

Abbildung der FSM auf ein digitales System (4)

```
module fsm_moore(   input    clk,in,reset_n,
                    output reg [3:0] out);
reg [1:0] next_state;
reg [1:0] state;
parameter zero=0, one=1, two=2, three=3;

// Next-State Logic
always @(*)
begin
        //code
end

// State-Update Logic
always @(posedge clk or negedge reset_n)
begin
        //code
end

// Output-Logic
always @(*)
begin
        //code
end
endmodule
```

```
// State-Update Logic
always @(       posedge clk or
                negedge reset_n)
begin
        //code
        if(!reset_n)
                state<=zero;
        else
                state<=next_state;
end
```

State-Update:
Bei jedem Takt wird der aktuelle „next-Zustand“ übernommen.

Abbildung der FSM auf ein digitales System (5)

```
module fsm_moore(   input    clk,in,reset_n,
                    output reg [3:0] out);
reg [1:0] next_state;
reg [1:0] state;
parameter zero=0, one=1, two=2, three=3;

// Next-State Logic
always @(*)
begin
        //code
end

// State-Update Logic
always @(posedge clk or negedge reset_n)
begin
        //code
end

// Output-Logic
always @(*)
begin
        //code
end
endmodule
```

```
// Output-Logic
always @(*)
begin
        //code
        case(state)
        zero:
                out<=4'b0001;
        one:
                out<=4'b0010;
        two:
                out<=4'b0100;
        three:
                out<=4'b1000;
        default:
                out<=4'b0001;
        endcase
end
endmodule
```

Ausgangswerte:
Zuweisung der festen Ausgangswerte anhand des derzeitigen Zustands.

Hinweis:
Da der Ausgang nur vom Zustand beeinflusst wird liegt hier eine Moore-FSM vor

Kompakte FSM-Struktur (Alternative)

```
module simple_fsm(input clk,in,reset_n, output reg [3:0] out);
reg [1:0] next_state;
reg [1:0] state;
parameter zero=0, one=1, two=2, three=3;
// All-Logic
always @(posedge clk or negedge reset_n)
begin
        if(!reset_n) begin
                        out<=4'b0001; state<=zero; end
        else
                case(state)
                zero: begin
                        out<=4'b0001; state<=one; end
                one: begin
                        out<=4'b0010; state<=in?zero:two; end
                two: begin
                        out<=4'b0100; state<=three; end
                three: begin
                        out<=4'b1000; state<=zero; end
                default: begin
                        out<=4'b0001; state<=zero; end
                endcase
end
endmodule
```

Beschreibung:

- Die Beschreibung ist nur noch sequentielle Logik
- Jeder Zustand definiert eigenständig seinen Ausgangszustand
- Jeder Zustand prüft eigenständig seinen Folgezustand