Digitale Schaltungstechnik, WS 2019/2020

9. Übungsblatt — Abgabe: 04.02.2020

1 Ampel

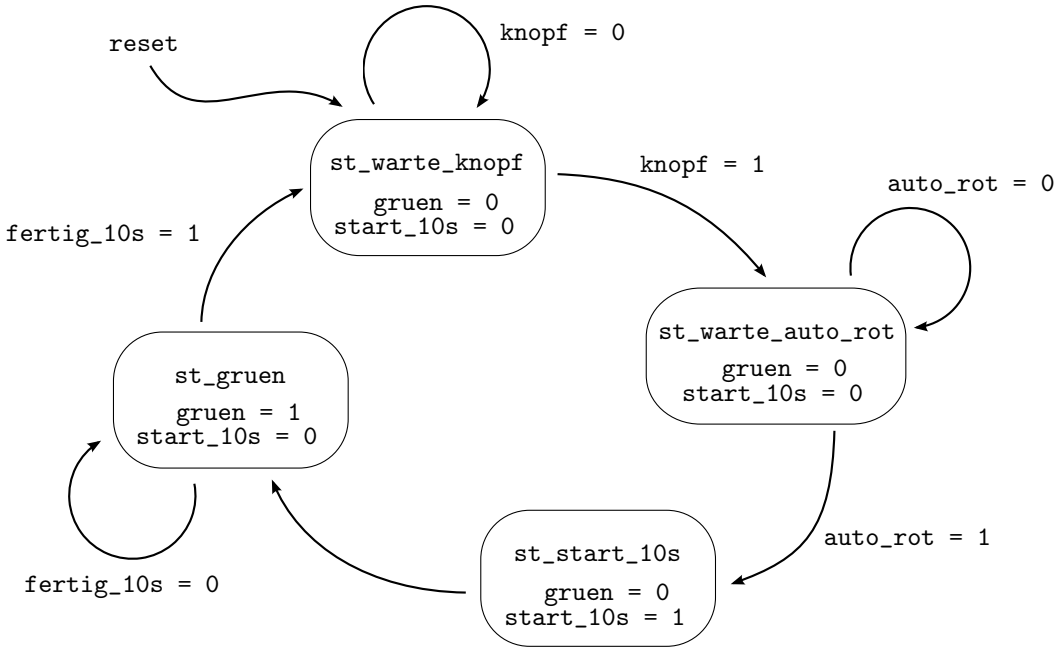
Als Beispiel für einen *endlichen Automaten* (*finite state machine*, kurz FSM) dient eine einfache Steuerung für eine Fußgängerampel. Die Ampel soll sich wie folgt verhalten:

- Wenn ein Fußgänger den Knopf drückt, wartet die Steuerung, bis die Ampel für die Autos auf „rot“ schaltet.
- Sobald die Autoampel rot wird, schaltet die Fußgängerampel auf „grün“.
- Nach zehn Sekunden wird die Fußgängerampel wieder rot und ist bereit für den nächsten Knopfdruck.

Dabei wird angenommen, dass es eine „Stoppuhr“ gibt, die von der Ampelsteuerung gestartet wird und ihr signalisiert, wenn zehn Sekunden vergangen sind.

Die Steuerung hat also (neben dem Takt- und Reset-Eingang) Eingänge für den Knopf, den die Fußgänger drücken (`knopf`), das Signal für die rote Autoampel (`auto_rot`) und das Signal der Stoppuhr (`fertig_10s`). Die Ausgänge der Steuerung sind das Startsignal für die Stoppuhr (`start_10s`) und natürlich das Signal für die grüne Fußgängerampel (`gruen`).

Den Automaten, der diese Steuerung implementiert, kann man durch das folgende Zustandsdiagramm mit den vier Zuständen `st_warte_knopf`, `st_warte_auto_rot`, `st_start_10s` und `st_gruen` darstellen:



Die Umsetzung als Verilog-Modul lautet:

```
module ampel_fsm
(
  input clk,
  input reset,
  input knopf,
  input auto_rot,
  input fertig_10s,
  output gruen,
  output start_10s
);

reg [2:0] current_state;
reg [2:0] next_state;

parameter st_warte_knopf    = 3'b000;
parameter st_warte_auto_rot = 3'b100;
parameter st_start_10s      = 3'b001;
parameter st_gruen          = 3'b010;
assign {gruen, start_10s} = current_state[1:0];

wire [2:0] inputvector;
assign inputvector = {knopf, auto_rot, fertig_10s};

always @(*) begin
  casex({inputvector, current_state})
    {3'b1xx, st_warte_knopf}:    next_state = st_warte_auto_rot;
    {3'b0xx, st_warte_knopf}:    next_state = st_warte_knopf;

    {3'bx1x, st_warte_auto_rot}: next_state = st_start_10s;
    {3'bx0x, st_warte_auto_rot}: next_state = st_warte_auto_rot;

    {3'bxxx, st_start_10s}:      next_state = st_gruen;

    {3'bxx1, st_gruen}:          next_state = st_warte_knopf;
    {3'bxx0, st_gruen}:          next_state = st_gruen;

    default: next_state = st_warte_knopf;
  endcase
end

always @(posedge clk or posedge reset) begin
  if (reset == 1'b1)
    current_state <= st_warte_knopf;
  else
    current_state <= next_state;
end

endmodule
```

a) Schauen Sie sich den Code an und versuchen Sie, den Zusammenhang zum Zustandsdiagramm nachzuvollziehen. Beachten Sie die folgenden verwendeten Techniken:

- Die drei Eingangssignale werden in der Variable `inputvector` zusammengefasst, die dann im `casex`-Block verwendet wird, um zusammen mit dem aktuellen Zustand `current_state` den nächsten Zustand `next_state` zu ermitteln.[1]
- Jeder Zustandsübergang, insbesondere das Verbleiben im selben Zustand, ist explizit im `casex`-Block angegeben. Der `default`-Fall sollte also normalerweise nie eintreten und dient nur dazu, dass die FSM wieder in einen definierten Zustand gelangt, falls das `current_state`-Register durch eine äußere Störung einen nicht als gültigen Zustand definierten Wert enthält.
- Die Kodierung der Zustände ist im Grunde beliebig, sie ist hier aber so gewählt, dass zwei Bits des Zustandsvektors direkt als Ausgang verwendet werden können.[2]
- Falls das `reset`-Signal aktiv ist, wird die FSM asynchron zum Takt in den angegebenen Anfangszustand versetzt. Ansonsten wird der Zustand in *jedem* Takt auf den im `casex`-Block ermittelten aktualisiert (hier zeigt sich wieder, dass auch das Verbleiben im selben Zustand als Zustandsübergang gilt).

b) Schreiben Sie ein Top-Level-Modul, in das Sie die FSM einbauen, um zu prüfen, ob sie richtig funktioniert. Die Signale `auto_rot` und `fertig_10s` können Sie der FSM durch das Drücken eines Knopfes geben und das Signal `start_10s` auf einer LED anzeigen (verwenden Sie ein langsames Taktsignal, damit Sie es auch sehen können).

c) *Bonusaufgabe:* Schreiben Sie ein Modul, das die Funktion der Stoppuhr übernimmt.

[1] In diesem `always`-Block wird das *blocking assignment* (`=`) verwendet, weil kombinatorische Logik beschrieben werden soll. Mehr Informationen zu diesem Thema gibt es unter `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.4021&rep=rep1&type=pdf`.

[2] Das Xilinx-Synthesewerkzeug XST wird die Zustände wahrscheinlich ohnehin wieder anders kodieren als Sie es im Code angeben. Näheres dazu erfahren Sie im XST User Guide (`http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/xst.pdf`) ab S. 203.