Thema 6: Statemaschine

Finite State Maschine

Folie 2

***

Zustandsmaschinen werden für Ansteuerung von digitalen Systemen verwendet. Man kann sie mit Programmen vergleichen, wo der Programmcode fest ist.

Die Ausgangssignale von Zustandsmaschinen, und allgemein von sequentiellen Schaltungen, hängen nicht nur von momentanen Werten der Eingangsvariablen sondern auch von deren Reihenfolge. Dieses Verhalten kann man nur dann realisieren wenn die Schaltung Speicherelemente enthält. Die Speicherelemente befinden sich in einem Zustand. Der nächste Zustand hängt vom momentanen Zustand und von den Eingangsvariablen.

Falls eine Statemaschine n Speicherelemente hat, kann sie sich theoretisch in 2^n verschiedenen Zustände befinden. Da es eine endliche Zahl von möglichen Zuständen gibt, nennt man solche Zustandsautomate „finite-state“ Maschinen.

Den Zustand des Speicherelements nennt man Zustandsvariable.

Folie 3

***

Die Zustandsmaschinen kann man in zwei Klassen unterteilen.

Beim Moore typ hängt der Ausgang der Zustandsmaschine nur von der Zustandsvariable – d.h. nur vom Zustand der Maschine.

Beim Mealy Typ hängt der Ausgang auch von den 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 aber oft mehr Zuständen.

Folie 4

***

Man kann in einem Zustandsautomat jede Art von Speicherzellen verwenden um den Zustand zu speichern.

Wenn alle Speicherzellen in 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.

Folie 5

***

Wir werden uns mit asynchronen Zustandsmaschinen befassen.

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.

Folie 6

***

Zustandsmaschine für einen Timer

Der Timer besteht aus folgenden Komponenten – einem Zähler, einem Komparator, einem Startkopf, einem Drehregler für die Zeiteinstellung und einem Lautsprecher/Klingel. 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.

Die Statmeschine braucht noch ein Taktsignal und ein asynchrons Reset.

7 ***

Entwurf von Zustandsautomaten ist schwer. Es verlangt Erfahrung und Übung

Zuerst kann man mit einem graphischen Zustandsdiagramm anfangen.

Wir schreiben grob die Zustände (oft wird am Anfang ein Zustand vergessen) und verbinden die Übergänge vom Zustand zum Zustand mit Pfeilen. Neben jedem Pfeil schreiben wir die Bedingung für den Übergang, also die Funktion der Eingangsvariablen die diesen Übergang ermöglich. Wenn es sich um eine synchrone Zustandsmaschine handelt (was wir annehmen), muss zuerst die Bedingung erfüllt werden, der Übergang passiert dann auf die nächste Taktflanke.

Z.B. der Zustandsautomat kommt von IDLE in RESETCNT Zustand auf die Taktflanke, wenn der Eingang Start aktiv (bzw. = 1) ist. Das zeigt das Zeitdiagramm.

Wer werden im weiterem Text die Taktflanken nicht explizit erwähnen: wenn wir sagen der Übergang passiert nachdem eine Bedingung erfüllt ist, meinen wir auf die nächste Taktflanke nach der erfüllten Bedingung.

Wir werden auch implizit annehmen, dass der Zustandsautomat in einem Zustand bleibt wenn die Bedingung für den Übergang nicht erfüllt ist. Man könnten dies auch mit schleifenförmigen Pfeilen beschrieben, wir werden es aber nicht tun, da es das Diagramm kompliziert macht.

Beachten wir, dass es Übergänge gibt die immer passieren, unabhängig von Eingangsvariablen. Diese werden mit „always“ gekennzeichnet. Bei den dazugehörigen Zuständen fehlen die Schleifen, da sie nie durchgeführt werden.

Ich werde hier auf einige Komplikationen eingehen.

8 ***

Nehmen wir an, es gibt im Diagramm eine Abzweigung. Aus einem Zustand kann der Automat in zwei verschiedenen Zuständen kommen. Wenn a = 1 ist der nächste Zustand A, wenn b=1 ist der nächste Zustand B. Hier stellt sich

Problem der Eindeutigkeit. Was passiert wenn beide Eingänge 1 sind, also a, b = 1, 1.

9***

Wir müssen eine Hierarchie der Eingänge definieren: z.B. a ist wichtiger als b. In dem Fall ist der nächste Zustand A, wenn a = 1, unabhängig vom b. Das dazugehörige Diagramm, ist in Folie gezeigt.

Beachten wir, dass solche Hierarchie der Bedingungen einfacher in HDL Sprache realisiert werden kann. Die Zustandsmaschine wird am einfachsten mit einem „CASE“ Befehl Beschrieben. Im Block „START“ werden alle Übergänge vom Zustand „START“ definiert. Der IF-ELSE Befehl definiert die Hierarchie. Der letzte ELSE Block wäre ausgeführt wenn weder a noch b wahr sind. In dem Fall bleibt der Zustand unverändert. Diese Zeile kann weggelassen werden, da Flipflops ihren Zustand behalten wenn es zu keiner Änderung kommt.

10***

Oft wird Reset-Eingang für den Zustandsautomat definiert. Wenn Reset = 1 ist, kommt der Automat in einen bestimmten Zustand. Das 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.

Nehmen wir im Moment an, es ist ein synchrones Reset und nach dem Reset kommt der Automat in Zustand START. Wir können das symbolisch wie auf der Folie definieren. Die eigentliche Realisierung ist komplizierter (s. Folie) – sie muss dafür sorgen dass Reset Priorität hat.

11***

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

Die weiteren Folien zeigen die Realisierung der Statemaschine des Timers.

Folien 12 - 20

***

Der Zustandsdiagramm könnte wie folgend aussehen (Code):

```
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
```

Die Folien zeigen auch die graphische Darstellung des Zustandsdiagrams.

Beachten wir, dass die Schleifen im Code weggelassen werden. Es wurde ein Asynchrones Reset verwendet.

Oft enthält der Code der Statemaschine auch die Digitalschaltungen, die die Statemaschine ansteuert. Das wäre in diesem Fall der Zähler selbst. Er ist im CASE Block COUNT.

Wir haben für die Zustände Gray Code verwendet da sich bei jedem Übergang nur ein Flipflop ändert was weniger Glitches verursacht. (Grey Code wird später erklärt.)

*** 21

Übungsaufgabe

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.

Der Zähler kann als eine komplexe Schaltung realisiert werden, die den Zähler selbst und die Statemaschine für die Ansteuerung enthält.

Der Zähler hat den Takteingang, einen enable - Eingang (CEn) und einen reset – Eingang (CRes). Der Zähler erzeugt den Ausgang „Wert“ – value und den Ausgang MaxCnt. MaxCnt = Maximalwert.

Das Top-Modul hat folgende Eingänge: clock, reset, enable, clear_overflow und die Ausgänge value, overflow und overflow_error.

Die Statemaschine verwendet 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 CEn sowohl von Zuständen als auch vom Eingang abgängig ist. Es handelt sich hier (in Bezug auf diese Ausgänge) um eine Mealy –

Zustandsmaschine. Eine Moor –Typ Realisierung würde mehr Zustände erfordern.

Der Code wird auf der Folie gezeigt.

```
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,

  output reg [WIDTH - 1:0] value_sr



);



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
```

Eine weitere einfachere Realisierung basiert auf zwei „Flags“ ERR und OVF.

```
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
```