

Ausgabe der Aufgabe: 27. September 2017

Eingabefrist: 23. Oktober 2017, 24:00

Hausaufgabe 1 kann bis zum Ende der Nachholungswoche (15. Dezember 2017, 24:00) mit der Einzahlung des Extraprozessengebührs eingegeben werden

Die Aufgabe wird in elektronischer Form, in PDF Format durch Hausaufgabeneingabeportal

<https://hf.mit.bme.hu/> eingegeben.

Registration ist erforderlich! Erreichbare Punkte: max. 15, IMSC Punktzahl: 5

Persönliche Digit-Kodes sind am Ende dieses Dokuments verfügbar!

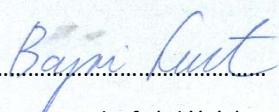
## Digitaltechnik Hausaufgabe 1

### Entwurf sequentieller Schaltnetze

<b>Name:</b> Bajcsi Levente	<b>NEPTUN:</b> XAO5ER	<b>Email:</b> levente.bajcsi@edu.bme.hu
<b>Lehrkreis:</b> Deutschkurs	<b>Jahrgang:</b> 2017	<b>DIGITCODE:</b> 2745136

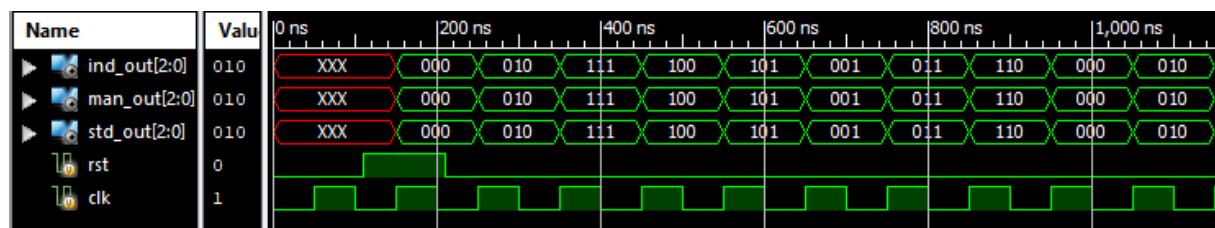
Die Aufgaben wurden selbständig, ohne unerlaubtes Hilfsmittel und ohne unmittelbare Mitwirkung der Anderen gelöst

(A feladatokat önállóan, meg nem engedett segédeszközök használatára és mások közvetlen közreműködése nélkül oldottam meg:)

  
Unterschrift (Aláírás)

### Zusammenfassung der Ergebnisse

Die drei Arten von endlichen Zustandsautomaten zählen nach der Reihenfolge, die durch meinen Digitcode angegeben wurde. Das gemeine Simulationsdiagramm sieht folgenderweise aus:



Anhand der Ergebnisse ist es klar, dass die Hausaufgabe gut gelöst wurde, weil die drei Ausgänge schauen die richtige Werte, was ich (anhand meines persönlichen Digit-Kodes) erwartete. Die drei verschiedene Weise, wodurch die Aufgabe gelöst wurde, waren alle sehr interessant zu entwickeln, aber am meisten gefällt mir die letzte, weil eine durch einer Binärzähler durchgeführte Lösung mich andernfalls nicht eingefallen hätte.

## HA1\_1

Die erste Weise war um eine Zustandsübergangstabelle orientiert. Meine Digit-Kode ergibt die folgende Tabelle:

Aktuell	Folgezustand		Aktuell	Folgezustand
0	2	→	0 – 000	2 – 010
2	7		1 – 001	3 – 011
7	4		2 – 010	7 – 111
4	5		3 – 011	6 – 110
5	1		4 – 100	5 – 101
1	3		5 – 101	1 – 001
3	6		6 – 110	0 – 000
6	0		7 – 111	4 – 100

Daraus die Karnaugh-Tabellen sind (Seien a, b, und c der dritten, zweiten und ersten Ziffern des aktuellen Zustandes in Binärform):

Folgezustand[0]:

	00	01	11	10
0	0	1	0	1
1	1	1	0	0

Folgezustand[1]:

	00	01	11	10
0	1	1	1	1
1	0	0	0	0

Folgezustand[2]:

	00	01	11	10
0	0	0	1	1
1	1	0	1	0

Die Ausgänge anhand dieser Tabellen sind:

$$\text{Folgezustand}[0] = /a */b *c + /a *b */c + a */b */c + a */b *c$$

$$\text{Folgezustand}[1] = /a */b */c + /a */b *c + /a *b *c + /a *b */c$$

$$\text{Folgezustand}[2] = /a *b *c + /a *b */c + a */b */c + a *b *c$$

Nach Minimisierung (zuerst durch die Einkreuzung von Primimplikanten):

$$\text{Folgezustand}[0] = a */b + /b *c + /a *b */c$$

$$\text{Folgezustand}[1] = /a$$

$$\text{Folgezustand}[2] = /a *b + b *c + a */b */c$$

Für wenigsten *product-terms* getestete Ausgänge, mithilfe von **Logic Friday**:

Minimized:

$$F0 = A B' C' + B C + A' B C';$$

$$F1 = A';$$

$$F2 = A B' C' + B' C + A' B C';$$

*Wenigste produkte*

Dieses Ergebnis zeigt, dass zwei Elemente können mehr als einmal verwendet werden ( $A*B*\bar{C}$  und  $\bar{A}*B*\bar{C}$ ). Diese Erscheinung bedeutet eine Verkleinerung in der Anzahl der nötigen Elemente in diesem Netzwerk.

Mit diesen logischen Ausdrücken habe ich die Module entwickelt, die bei jeder steigenden Taktflanke den nächsten Zustand (für jedes Bit) anhand des aktuellen Zustandes berechnet. Diese Implementation folgt den Moore-Modell, weil die Ausgänge nur von dem aktuellen Zustand abhängen.

In der Implementation bei jeder steigenden Taktflanke wird jedes Bit anhand dieser Formeln berechnet (oder bei RESET wird jedes Bit nulliert), dann zu den Ausgängen ausgeführt.

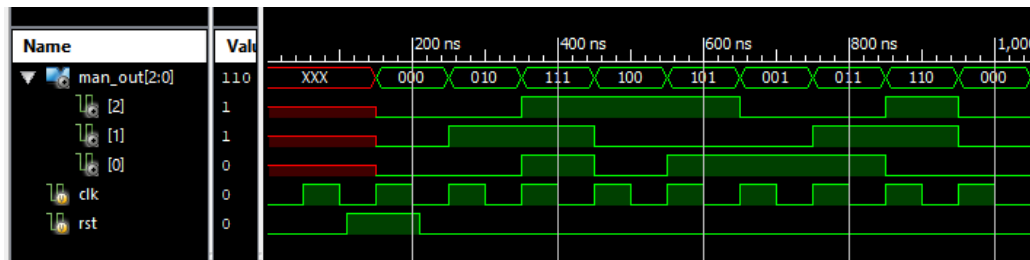
Die Code meiner Implementation ist hier zu sehen:

```

21 module MAN_FSM(
22     input clk,
23     input rst,
24     output [2:0] man_out
25 );
26
27     reg [2:0] state;
28     reg [2:0] next_state;
29     wire a, b, c;
30     assign {a,b,c} = state;
31
32     always @ (posedge clk)
33     begin
34         if(rst) state <= 3'b0;
35         else
36             begin
37                 next_state[0] = a & ~b & ~c | ~b & c | ~a & b & ~c;
38                 next_state[1] = ~a;
39                 next_state[2] = a & ~b & ~c | b & c | ~a & b & ~c;
40                 state <= next_state;
41             end
42     end
43
44     assign man_out = state;
45
46 endmodule

```

Das Ergebnis der Simulation:

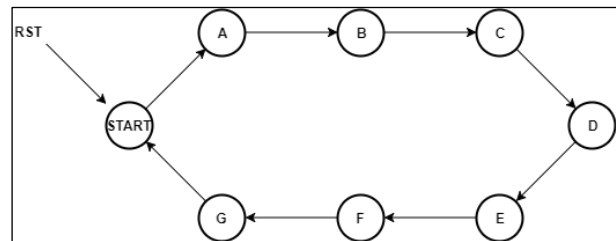


## HA1\_2

Die zweite Weise wurde durch dem Allgemeinen FSM-Planungsmethodik durchführbar (mit den Schritten angeschaut):

1. Die Zustände definieren:

START	0 – 000
A	2 – 010
B	7 – 111
C	4 – 100
D	5 – 101
E	1 – 001
F	3 – 011
G	6 – 110



2. Entscheidung darüber, dass ich den **Moore-Modell** brauche (ähnlich zu HA1\_1, nur der aktuelle Zustand soll die Ausgänge beeinflussen).
3. Der Zustandsdiagramm zeigt, wie diese Zustände einander folgen sollten.

Mithilfe von diesen Punkten die Entwicklung des Modules war eigentlich einfach: für jeden Zustand definierte ich einen Parameter, dessen Name START für 0, und A-G für die Reste war (Sehen Sie die Tabelle oben für die Zuordnungen).

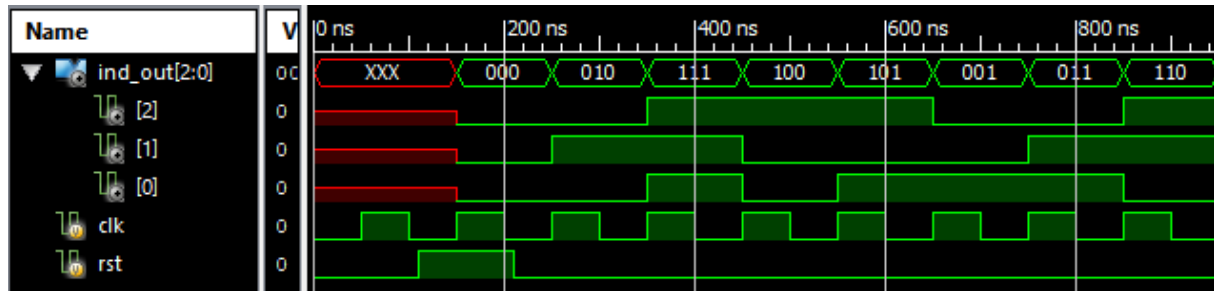
Bei jeder steigenden Taktflanke wird der Zustand aktualisiert mit dem Wert von dem nächsten Zustand, der bei anhand der Zustandsdiagram „berechnet“ wird (bei RESET wird START als Zustand ausgegeben).

Die Code meiner Implementation ist hier zu sehen:

```

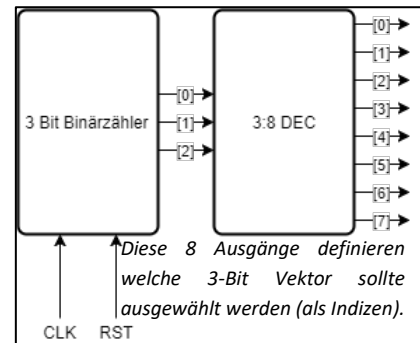
21 module STD_FSM(
22     input clk,
23     input rst,
24     output [2:0] std_out
25 );
26     parameter START = 3'b000;
27     parameter A = 3'b010;
28     parameter B = 3'b111;
29     parameter C = 3'b100;
30     parameter D = 3'b101;
31     parameter E = 3'b001;
32     parameter F = 3'b011;
33     parameter G = 3'b110;
34
35     reg [2:0] state;
36     reg [2:0] next_state;
37     assign std_out = state;
38
39     always @ (posedge clk)
40     begin
41         if(rst) state <= START;
42         else state <= next_state;
43     end
44
45     always @ (*)
46     begin
47         case(state)
48             START: next_state <= A;
49             A: next_state <= B;
50             B: next_state <= C;
51             C: next_state <= D;
52             D: next_state <= E;
53             E: next_state <= F;
54             F: next_state <= G;
55             G: next_state <= START;
56         endcase
57     end
58 endmodule
  
```

Das Ergebnis der Simulation:



### HA1\_3

Die dritte Weise ähnelt die zweite darin, dass die Zustände nach ihren „Indizes“ behandelt werden, aber anstatt Parametern (START, A-G) benutzen wir jetzt einen 3:8 Dekoder in einem Busmultiplexer, anhand dessen und eines Binärzählers (der die Eingänge von dem Dekoder generiert) den Wert von dem nächsten Index berechnet und zu den Ausgängen ausgeführt wird. (Zum Beispiel, wenn der Binärzähler den Wert  $101_2$  nach den Dekoder „schickt“, wählt die Dekoder den  $101_2$ -ten (5-ten) Index aus, und deshalb wird der aktuelle Zustand 1 als Wert haben):



Die Implementation dieses Modules folgt diese Logik Schritt für Schritt: ein *always*-Block zählt in 3 Bits von 0 bis 7 bei jeder steigenden Taktflanke (dann wegen Überlauf wieder von 0) und ein anderer ist der Busmultiplexer, dessen Eingänge die Ausgänge des Binärzählers sind und als Ausgang gibt den Wert des aktuellen Zustands aus (mit einer *case*-Statement in Verilog durchgeführt). RST nulliert die Zahl in dem Binärzähler.

Die Code meiner Implementation ist hier zu sehen:

```

21 module IND_FSM(
22     input clk,
23     input rst,
24     output [2:0] ind_out
25 );
26
27     reg [2:0] out, cnt;
28     assign ind_out = out;
29
30     always @ (posedge clk)
31     begin
32         if(rst) cnt <= 3'o0;
33         else cnt <= cnt + 3'o1;
34     end
35
36     always @ (*)
37     begin
38         case(cnt)
39             3'o0: out <= 3'b000;
40             3'o1: out <= 3'b010;
41             3'o2: out <= 3'b111;
42             3'o3: out <= 3'b100;
43             3'o4: out <= 3'b101;
44             3'o5: out <= 3'b001;
45             3'o6: out <= 3'b011;
46             3'o7: out <= 3'b110;
47         endcase
48     end
49 endmodule

```

Der Topmodule, die diese Einheiten zusammenhielt, instanziiert jede Submodule, damit sie funktionellen können. Die Code dafür:

```

5 module HF1(
6     input rst,
7     input clk,
8     output [2:0] ind_out,
9     output [2:0] man_out,
10    output [2:0] std_out
11 );
12
13     IND_FSM ind(.rst(rst), .clk(clk), .ind_out(ind_out));
14     MAN_FSM man(.rst(rst), .clk(clk), .man_out(man_out));
15     STD_FSM std(.rst(rst), .clk(clk), .std_out(std_out));
16
17 endmodule
18

```

Das Ergebnis der Simulation:

