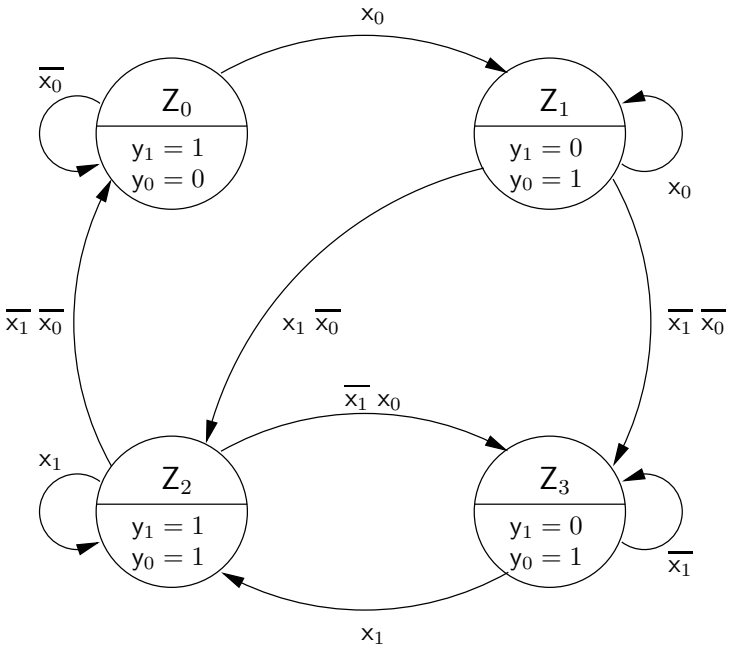
64-041 Übung Rechnerstrukturen und Betriebssysteme

Aufgabenblatt 10 Ausgabe: 15.12., Abgabe: 05.01. 24:00

Gruppe	
Name(n)	Matrikelnummer(n)

Aufgabe 10.1 (Punkte 10+10+10)

Schaltwerk-Analyse: Wir betrachten das Zustandsdiagramm eines Moore-Schaltwerks mit Eingängen $X(x_0, x_1)$ und Ausgaben $Y(y_0, y_1)$ sowie vier Zuständen Z_0, Z_1, Z_2, Z_3. Die Zustände $Z\ (z_1, z_0)$ werden binär durchgezählt: $Z_0 = (0,0)$, $Z_1 = (0,1)$, $Z_2 = (1,0)$ und $Z_3 = (1,1)$.



(a) Ermitteln Sie aus dem Zustandsdiagramm die minimierten Ausdrücke für die Zustandsübergangsfunktion δ, die den Folgezustand Z^+ aus aktuellem Zustand Z und den Eingabewerten X berechnet.

(b) Ermitteln Sie aus dem Zustandsdiagramm die minimierten Ausdrücke der Ausgangsfunktion λ zur Berechnung des Ausgangsvektors Y.

(c) Überprüfen Sie den Automaten auf Vollständigkeit (in jedem Zustand ist für jede Eingangsbelegung mindestens ein Übergang aktiv) und Widerspruchsfreiheit (in jedem Zustand ist für jede Eingangsbelegung höchstens ein Übergang aktiv).

Aufgabe 10.2 (Punkte 10+10+10)

Entwurf eines Schaltwerks: Wir betrachten ein Schaltwerk mit sechs Zuständen $Z_0, \ldots Z_5$, einem Eingang x und vier Ausgängen y_3, y_2, y_1, y_0. Die Zustandsübergänge und die Ausgabe sind in der Tabelle angegeben:

x	Z	Z^+	y_3	y_2	y_1	y_0
0	Z_0	Z_1	0	1	0	0
0	Z_1	Z_2	0	0	0	0
0	Z_2	Z_3	1	0	0	0
0	Z_3	Z_5	0	1	1	1
0	Z_4	Z_0	0	0	1	0
0	Z_5	Z_4	1	1	0	1
1	Z_0	Z_2	0	1	0	0
1	Z_1	Z_4	0	0	0	0
1	Z_2	Z_0	1	0	0	0
1	Z_3	Z_4	0	1	1	1
1	Z_4	Z_3	0	0	1	0
1	Z_5	Z_2	1	1	0	1

(a) Zeichnen Sie das Zustandsdiagramm des Schaltwerks.

(b) Um das Schaltwerk zu realisieren, wählt man jetzt eine Codierung der Zustände. Wir betrachten zwei (von vielen) Möglichkeiten.
Bestimmen Sie für beide Codierungen die Funktionen des Zustandsübergangsschaltnetzes (das δ-Schaltnetz). Beachten Sie dabei die Möglichkeit von *Don't-Cares*. Die Tabellen und KV-Diagramme sollen mit abgegeben werden.

Z_i	Codierung 1 $(z_2\, z_1\, z_0)$	Codierung 2 $(z_2\, z_1\, z_0)$
Z_0	(100)	(001)
Z_1	(000)	(110)
Z_2	(011)	(000)
Z_3	(001)	(101)
Z_4	(010)	(100)
Z_5	(101)	(010)

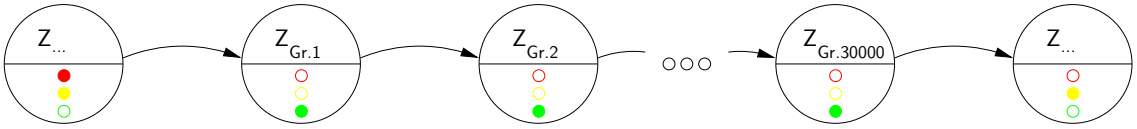
(c) Offenbar lässt sich durch eine geeignete Codierung der Zustände eine erhebliche Vereinfachung der Schaltfunktionen erreichen. Das Problem ist nur, dass es alles andere als einfach ist, eine bestmögliche Codierung zu finden, wobei man dann auch noch die Funktionen für die Ausgabe (das λ-Schaltnetz) mit berücksichtigen müsste.

Geben Sie eine Codierung für die sechs Zustände an, die zumindest das λ-Schaltnetz des Automaten minimiert. Es sind dabei auch mehr als drei Bits für die Zustandscodes erlaubt. Erläutern Sie ihre Vorgehensweise.

Aufgabe 10.3 (Punkte 5+10+10)

Gekoppelte Automaten: In der Vorlesung wurde eine Ampelschaltung (Folie 697ff.) vorgestellt. Wird das System jetzt aber mit einem konstanten Takt von beispielsweise 1 KHz betrieben, dann muss man für einen realistischen Betrieb die Zeiten der jeweiligen Ampelphasen deutlich verlängern. Soll beispielsweise die Grünphase 30 Sekunden dauern, dann entspricht das 30 000 Takten.

(a) Eine ad-hoc Lösung wäre natürlich, wie in der Grafik unten skizziert, entsprechend viele Zustände einzuführen, die nacheinander durchlaufen werden.



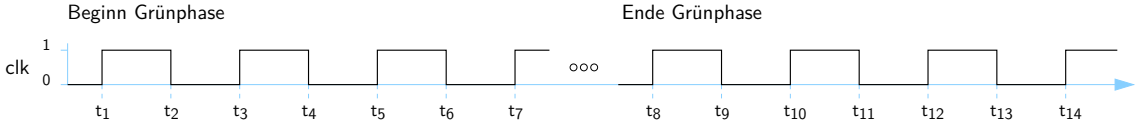
Dieser Ansatz hat jedoch mehrere Nachteile, weswegen man üblicherweise zwei (oder mehr) *gekoppelte Automaten* einsetzt: einen „Hauptautomaten" der die Zustandsübergänge realisiert und einen Zähler (trivialer Automat), der für die Wartezeiten sorgt. Zählen Sie kurz auf (jeweils ein Satz zur Begründung), welche Nachteile dies sein könnten.

(b) Der Zähler ist hier beispielhaft mit der Hardwarebeschreibungssprache VHDL dargestellt und arbeitet folgendermaßen:

- Ein asynchrones Reset-Signal (= unabhängig vom Takt) `rst` setzt den Zähler auf 0.
- Mit der Taktvorderflanke (`rising_edge`) ändert sich der Zustand/Zählerstand.
- Bei aktivierter Steuerleitung `load` wird der Zähler mit dem Startwert `start` initialisiert. Ansonsten wird rückwärts gezählt, solange der Zählerstand > 0 ist.
- Als Ausgabe gibt die Leitung `isZero` an, ob der Zählerstand 0 erreicht wurde.

```
1  -- countDown.vhd
2  ----------------------------------------------------------------------
3  library IEEE;
4  use IEEE.numeric_bit.all;
5
6  entity countDown is
7  generic(bitWd           : positive := 16);
8  port(   clk, rst, load  : in  bit;
9          start           : in  integer range 0 to (2**bitWd)-1;
10         isZero          : out bit);
11 end entity countDown;
12
13 architecture behaviour of countDown is
14   signal cnt    : integer range 0 to (2**bitWd)-1; -- 'Zählvariable'
15 begin
16   cnt_P: process (rst, clk) is
17   begin
18     if rst = '1' then           cnt <= 0;       -- async. Reset: 0
19     elsif rising_edge(clk) then                 -- Taktvorderflanke
20       if load = '1' then        cnt <= start;   --   Startwert laden
21       elsif cnt > 0 then        cnt <= cnt-1;   --   Runterzählen
22       end if;
23     end if;
24   end process cnt_P;
25
26   isZero <= '1' when cnt = 0 else '0';          -- Ausgang: isZero?
27 end architecture behaviour;
```

Der Automat für die Ampel sei auch mit vorderflankengesteuerten Flipflops im Zustandsregister Z realisiert. Beschreiben Sie textuell, wie sich die Signale `load`, `start` und `isZero` zu Beginn und zum Ende der Grünphase (Zeitpunkte: $t_1 \ldots t_{14}$) verhalten.



Tipp: beim Taktschema gekoppelter Automaten ist entscheidend, wann die einzelnen Zustandsübergänge stattfinden. Durch die gegenseitigen Abhängigkeiten (Steuersignale und Zählerstände) kann es leicht vorkommen, dass man einen Takt „zu spät“ ist.

(c) Was würde passieren, wenn die Automaten mit unterschiedlichen Taktflanken arbeiten: die Ampelsteuerung mit der Taktvorderflanke und der Zähler mit der Rückflanke. Verfahren Sie wie im vorigen Aufgabenteil (b).

Aufgabe 10.4 (Punkte 10+5)

Installation und Test der GNU Toolchain: In Vorbereitung auf Kapitel 13 zum x86-Assembler und analog zu den Beispielen in Kapitel 2 ab Folie 90, sollen Sie selbst Zugang zu einem C-Compiler und den zugehörigen Tools haben. Wir empfehlen die *GNU Toolchain* mit dem `gcc` C-Compiler und Werkzeugen. Diese ist auf den meisten Linux-Systemen bereits vorinstalliert, so dass Sie die Befehle direkt ausführen können.

Normalerweise könnten Sie dazu die Dual-boot Rechner in den PC-Poolräumen nutzen... – was natürlich den Zugang zum Informatikum voraussetzt. Wenn Sie zu Hause lernen und keinen Linux Rechner haben, bzw. nicht die Cygwin-Umgebung (s.u.) installieren wollen, können sie sich auch per `ssh` auf der `rzssh1.informatik.uni-hamburg.de` einloggen und die dort installierten Programme nutzen.

Hauptsächlich soll Sie die Aufgabe dazu motivieren vielleicht auf dem eigenen Rechner mit den Werkzeugen zu „spielen“ und so auch selbst zu sehen was aus programmiertem Code auf niedrigeren Abstraktionsebenen wird. Also installieren Sie die entsprechenden Programme (distributionsabhängig).

Für Windows-Systeme könnten Sie die Cygwin-Umgebung von `cygwin.com` herunterladen und installieren. Im Setup von Cygwin dann bitte den `gcc`-Compiler und die Entwickler-Tools auswählen und installieren. Alternativ können Sie auch einen anderen C-Compiler verwenden, Sie müssen sich dann aber die benötigten Befehle und Optionen selbst heraussuchen.

Anmerkung: Keine Angst, die Aufgabe soll zeigen, wie Assemblercode aussieht und Ihnen helfen erste Einblicke zu gewinnen, wie Betriebssystem, (Programm-) Binär-Code und die Hardware zusammenspielen. **Es geht nicht darum Assemblerprogrammierung zu lernen!**

Für einen ersten Test tippen Sie bitte das folgenden Programm ab oder laden Sie sich die Datei `aufg10_4.c` herunter. Passen Sie die Datei an, indem Sie dort ihren Namen und die Matrikelnummer eintragen. Anschließend sollen Sie das Programm übersetzen und sich den erzeugten Assembler- und Objektcode analysieren.

```c
/* aufg10_4.c
 * Einfaches Programm zum Test des C-Compilers und der zugehörigen Tools.
 * Bitte setzen Sie in das Programm ihren Namen und die Matrikelnummer ein
 * und probieren Sie alle der folgenden Operationen aus:
 *
 * Funktion         Befehl                                  erzeugt
 * ----------------+---------------------------------------+----------------
 * C -> Assembler:  gcc -O2 -S aufg10_4.c                  -> aufg10_4.s
 * C -> Objektcode: gcc -O2 -c aufg10_4.c                  -> aufg10_4.o
 * C -> Programm:   gcc -O2 -o aufg10_4.exe aufg10_4.c     -> aufg10_4.exe
 * Disassembler:    objdump -d aufg10_4.o
 *                  objdump -d aufg10_4.exe
 * Ausführen:       aufg10_4.exe
 */

#include <stdio.h>

int main( int argc, char** argv )
{ int matrikelNr    = 123456;
  char vorname[32]  = "John";
  char nachname[32] = "Doe";
//char *vorname     = "John";
//char *nachname    = "Doe";

  printf("Name: %s %s - Matrikelnr.: %d\n", vorname, nachname, matrikelNr);
  return 0;
}
```

(a) Machen Sie sich mit dem Compiler und den Tools vertraut. Probieren Sie die vorgeschlagenen Befehle aus und sehen Sie sich die Ausgaben an.

Erzeugen Sie eine Textdatei, die die Ausgabe des Programms und ein Listing des Disassemblers enthält. Dies geschieht am einfachsten mit den folgenden Befehlen:

```
./aufg10_4.exe         > loesung10_4.txt
echo "=========================================" >> loesung10_4.txt
objdump -d aufg10_4.o >> loesung10_4.txt
```

Markieren Sie in der Datei an welcher Stelle des Codes: Vorname, Nachname und Matrikelnummer stehen (mit kurzer Begründung). Diese Datei ist als Lösung des Aufgabenteils abzugeben.

(b) In dem Code `aufg10_4.c` sind die Zeilen 22 und 23 auskommentiert. Ändern Sie die Variablen für Vor- und Nachnamen in die zweite Version (Zeiger auf den String, statt `char`-Array).

Was ändert sich in dem Assembler-Code? Es genügt, die Änderungen (inhaltlich) zu beschreiben, es müssen keine Listings abgegeben werden.