64-041 Übung Rechnerstrukturen

Aufgabenblatt 11 Ausgabe: 20.12.13, Abgabe: 10.1.14 12:00

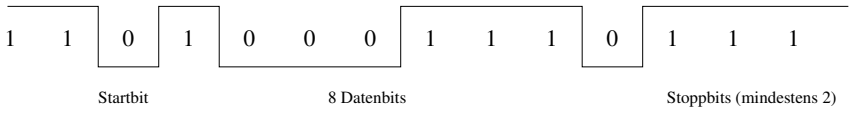
Gruppe	
Name(n)	Matrikelnummer(n)

Aufgabe 11.1 (Punkte 25)

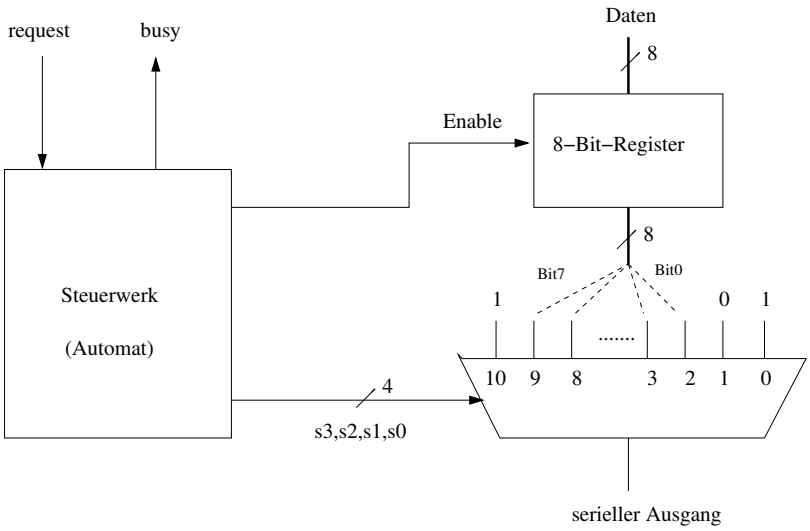
Serielle Schnittstelle (RS-232): Früher hatte jeder PC mindestens eine sog. Serielle Schnittstelle, mit der sich Daten zwischen zwei Geräten übertragen ließen. Auch heute arbeiten noch Bluetooth-, Midi- und Messgeräte mit (virtuellen) TTYs. Wir wollen uns hier das üblicherweise verwendete Protokoll ansehen. Für weitere Einzelheiten sei dabei auf Wikipedia oder andere Quellen im Netz verwiesen.

Übertragen wurde asynchron, d.h. Sender und Empfänger müssen sich zunächst über die sog. Baudrate, die Zahl der Bits pro Sekunde, die übertragen werden soll, einig sein. Üblich waren Übertragungsraten von 2400, 4800, 9600 und 19200 Baud. Inzwischen lassen sich auch viel höhere Werte erreichen.
Wenn der Sender keine Daten zu senden hat, legt er eine **1** auf die Leitung. Die Übertragung eines Zeichens (8 Bit) beginnt dann mit einer **0**, dem sog. Startbit. Dann werden nacheinander die 8 Bits des Zeichens übertragen, wobei mit dem niederwertigen Bit gegonnen wird. Es folgt dann eine Anzahl von **1**-Bits (sog. Stoppbits), über die sich Sender und Empfänger ebenfalls einig sein sollten. Üblich sind ein oder zwei. Wenn mehr Einsen als Stoppbits gesendet werden, ist das unkritisch, nur weniger bringen das Protokoll durcheinander.

Sehen wir uns ein Beispiel an. Wir wollen den Buchstaben G (im ASCII-Code 71_{16}) über eine serielle Schnittstelle mit übertragen und haben zwei Stoppbits festgelegt. Das sieht dann so aus:



Ihre erste Aufgabe ist es nun ein Schaltwerk zu entwerfen, das einen Wert (Buchstaben) in dem oben angegeben Protokoll (8 Datenbit und zwei Stoppbit) auf einer seriellen Schnittstelle ausgibt. Genauer soll in der angegeben Schaltung das Steuerwerk realisiert werden.



Dazu noch einige Erläuterungen:

Das Signal *busy* sagt der übergordneten Einheit, dass der Automat noch damit beschäftigt ist, ein Zeichen auf der seriellen Schnittstelle auszugeben. Nur wenn *busy 0* ist, lohnt es sich, über das Signal *request* den Automaten dazu aufzufordern, mit der Ausgabe eines weiteren Zeichens zu beginnen.
Das Signal *Enable* sagt dem Register, dass mit der nächsten Taktflanke ein neuer Wert eingelesen werden soll. (**1** - einlesen eines neuen Wertes).
Die Signale *s3, s2, s1, s0* dienen zur Ansteuerung des 11-1-Multiplexers. Für 1000 wird z.B. der Eingang 8 durchgeschaltet.

Wenn sich der Automat im Startzustand befindet und das Signal *request* **0** ist, bleibt der Automat dort und auf der Schnittstelle wird eine **1** ausgegeben.
Wenn sich der Automat im Startzustand befindet und das Signal *request* **1** ist, heißt das, dass auf der seriellen Schnittstelle ein neuer Wert ausgegeben werden soll, der von der übergordneten Einheit am Eingang des Registers bereitgestellt wird. Der Automat soll dann mit seiner Arbeit beginnen, also *busy* auf **1** setzen und auf der Schnittstelle erst eine **0** als Startbit ausgeben, danach die 8 Datenbits und schließlich eine **1** als Stoppbit, um dann wieder in den Startzustand zu gehen. Wenn sich der Automat nicht im Startzustand befindet, sollte er nicht auf das Eingabesignal *request* reagieren.

All das lässt sich mit einem Automaten realisieren, der 11 Zustände hat.

(a) Zeichnen Sie das Zustandsdiagramm des Automaten.

(b) Vervollständigen Sie die Zustandstabelle des Automaten, indem Sie die fehlenden Zustände und die zugehörigen Ausgangswerte ergänzen. Die Tabelle enthält links den Eingangswert *request* und den aktuellen Zustand Z in 4-bit Binärcodierung (z_3, z_2, z_1, z_0). Als Zustandscodierund wählen wir 0000 für den Startzustand und dann fortlaufend 0001 bis 1010. Angegeben sind dann der Folgezustand Z^+ und die Ausgangswerte zum Ansteuern der Signale *busy*, *Enable* und *s3,s2,s1,s0*.

req.	z_3	z_2	z_1	z_0	z_3^+	z_2^+	z_1^+	z_0^+	busy	Ena.	s3	s2	s1	s0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
$*$	0	0	0	1										
		...								...				
$*$	1	1	1	1										

(c) Übertragen Sie die zehn boolschen Funktionen aus der obigen Zustandstabelle in KV-Diagramme und minimieren Sie sie. Markieren Sie mögliche Schleifen und geben Sie die zugehörigen Ausdrücke für den Folgezustand (z_3^+, z_2^+, z_1^+, z_0^+) und die Ausgabewerte *busy*, *Enable*, *s3,s2,s1,s0* in disjunktiver Form an.
Achtung: Das Signal *request* brauchen Sie dabei **nicht** zu berücksichtigen, da es sich sowieso nur auf die Funktion z_0^+ auswirkt und vor allem, weil Sie in der Vorlesung keine KV-Diagramme mit mehr als vier Eingabevariablen kennengelernt haben. Die endgültige Funktion für z_0^+ würde lauten $z_0^+ = request\,\overline{z_3}\,\overline{z_0} \vee F(Z)$, wobei $F(Z)$ die Funktion ist, die Sie aus ihrem KV-Diagramm für z_0^+ erhalten, wenn Sie als Folgeszustand von 0000 immer 0000 annehmen.

Aufgabe 11.2 (Punkte 10+10)

Installation und Test der GNU-Toolchain: Ziel dieser Aufgabe ist es, dass 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.

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

Als dritte Alternative können Sie auf die Rechner in den PC-Poolräumen unseres RZ zurückgreifen, die PCs sind als Dual-Boot Systeme auch mit einer Linux Distribution (Ubuntu 12.04) ausgestattet.

Für einen ersten Test tippen Sie bitte das folgenden Programm ab oder laden Sie sich die Datei `aufg11_2.c` von der Webseite herunter. Ändern Sie dann die Datei, indem Sie ihre eigene Matrikelnummer eintragen. Übersetzen Sie das Programm und schauen Sie sich den erzeugten Assembler- und Objektcode an.

```
/* aufg11_2.c
 * Einfaches Programm zum Test des gcc-Compilers und der zugehörigen Tools.
 * Bitte setzen Sie in das Programm ihre Matrikelnummer ein und probieren
 * Sie alle der folgenden Operationen aus:
 *
 * Funktion          Befehl                                  erzeugt
 * ------------------+---------------------------------------+-----------------
 * C -> Assembler:   gcc -O2 -S aufg11_2.c                   -> aufg11_2.s
 * C -> Objektcode:  gcc -O2 -c aufg11_2.c                   -> aufg11_2.o
```

```
 * C -> Programm:    gcc -O2 -o aufg11_2.exe aufg10_2.c   -> aufg11_2.exe
 * Disassembler:     objdump -d aufg11_2.o
 * Ausführen:        aufg11_2.exe
 *
 * 32bit Code auf 64bit System: gcc -m32 ...
 */

#include <stdio.h>

int main( int argc, char** argv )
{ int matrikelNr = 123456;

  printf( "Meine Matrikelnummer ist %d (0x%x)\n", matrikelNr, 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.

Hinweis: Auf x86-64 Systemen (64bit Linux) können Sie auch die `gcc`-Compileroption `-m32` ausprobieren, um 32bit Code zu erzeugen.

(b) Schicken Sie den Quellcode sowie den erzeugten Assemblercode und die Ausgabe des Befehls `objdump -d` (GNU Toolchain) an Ihren Gruppenleiter.

Bei Verwendung anderer Compiler und Tools bitte ebenfalls die entsprechenden Ausgabedateien generieren und einschicken.

Hinweis: Der erzeugte Programmcode (`aufg11_2.exe`) muss nicht mit abgegeben werden. Verschiedene Mailserver halten Mails mit angehängten ausführbaren Programmen wegen eventuell enthaltener Viren automatisch zurück.

Aufgabe 11.3 (Punkte 3+3+3+3+3)

Adressierung: Auf einer 1-Adress Maschine (Akkumulatormaschine) werden Ladebefehle mit unterschiedlichen Adressierungsmodi ausgeführt. Der Speicher enthält folgende Werte:

Adresse	Inhalt
20	60
30	40
40	20
50	40
60	20
70	80
80	50

Welcher Wert steht jeweils nach Ausführung der folgenden Befehle im Akkumulator?

(a) `LOAD IMMEDIATE 30`

(b) `LOAD DIRECT 30`

(c) `LOAD INDIRECT 30`

(d) `LOAD DIRECT 50`

(e) `LOAD INDIRECT 80`

Aufgabe 11.4 (Punkte 5·8)

Befehlsformate: Vergleichen Sie 0-, 1-, 2- und 3-Adress Maschinen, indem Sie für jede Architektur ein Programm zur Berechnung des folgenden Ausdrucks schreiben:

$$R = (A * B - C)/(D + E * F)$$

Für die unterschiedlichen Maschinentypen sind die jeweils verfügbaren Befehle unten angegeben. Bezeichner M und N stehen für 16-bit Speicheradressen und MEM[M] ist der Inhalt des Speichers an der Adresse M. Mit X, Y und Z werden 4-bit Registernummern codiert.

0-Adress Maschine mit einen unbegrenzten Stack (TOS "top of stack")

Mnemonik	Bedeutung
`PUSH M`	push; TOS = MEM[M]
`POP  M`	MEM[M] = TOS; pop
`ADD`	tmp = TOS; pop; TOS = tmp + TOS
`SUB`	tmp = TOS; pop; TOS = tmp − TOS
`MUL`	tmp = TOS; pop; TOS = tmp ∗ TOS
`DIV`	tmp = TOS; pop; TOS = tmp / TOS

1-Adress Maschine: Akkumulatormaschine mit genau einem Register

Mnemonik	Bedeutung
`LOAD  M`	Akku = MEM[M]
`STORE M`	MEM[M] = Akku
`ADD M`	Akku = Akku + MEM[M]
`SUB M`	Akku = Akku − MEM[M]
`MUL M`	Akku = Akku ∗ MEM[M]
`DIV M`	Akku = Akku / MEM[M]

2-Adress Maschine: benutzt nur Speicheroperanden

Mnemonik	Bedeutung
`MOV M,N`	MEM[M] = MEM[N]
`ADD M,N`	MEM[M] = MEM[M] + MEM[N]
`SUB M,N`	MEM[M] = MEM[M] − MEM[N]
`MUL M,N`	MEM[M] = MEM[M] ∗ MEM[N]
`DIV M,N`	MEM[M] = MEM[M] / MEM[N]

3-Adress Register-Maschine: *load-store* RISC-Architektur, 16 Universalregister

Mnemonik	Bedeutung
`LOAD X,M`	X = MEM[M]
`STORE M,X`	MEM[M] = X
`MOV X,Y`	X = Y
`ADD X,Y,Z`	X = Y + Z
`SUB X,Y,Z`	X = Y − Z
`MUL X,Y,Z`	X = Y ∗ Z
`DIV X,Y,Z`	X = Y / Z

(a) Schreiben Sie vier (möglichst kurze) Programme für die Berechnung des Ausdrucks $R = (A * B - C)/(D + E * F)$ auf den verschiedenen Maschinen. Dabei bedeuten $A \ldots F$ und R die Speicheradressen der Operanden bzw. des Resultats. Verwenden Sie, falls nötig, die Speicheradressen von $G \ldots Q$ für Zwischenergebnisse.

(b) Wenn die Befehlscodierung jeweils 8-bit für den Opcode verwendet (und natürlich 16-bit für eine Speicheradresse bzw. 4-bit für eine Registernummer), wie viele Bits werden dann für jedes der obigen vier Programme benötigt?

Welche Maschine hat also die kompakteste Codierung (gemessen an der Programmgröße in Bits) für dieses Programm?