64-041 Übung Rechnerstrukturen

Aufgabenblatt 10 Ausgabe: 17.12.14, Abgabe: 7.1.15 24:00

Gruppe	
Name(n)	Matrikelnummer(n)

Aufgabe 10.1 (Punkte 25)

Entwurf eines Schaltwerks
Wir betrachten ein Schaltwerk mit sechs Zuständen $s_0, \ldots s_5$, einem Eingang x und vier Ausgängen y_1, y_2, y_3, y_4. Die Zustandsübergänge und die Ausgabe sind dabei durch folgende Tabelle gegeben:

x	S	S^+	y_1	y_2	y_3	y_4
0	s_0	s_1	0	1	0	0
0	s_1	s_2	0	0	0	0
0	s_2	s_3	1	0	0	0
0	s_3	s_5	0	1	1	1
0	s_4	s_0	0	0	1	0
0	s_5	s_4	1	1	0	1
1	s_0	s_2	0	1	0	0
1	s_1	s_4	0	0	0	0
1	s_2	s_0	1	0	0	0
1	s_3	s_4	0	1	1	1
1	s_4	s_3	0	0	1	0
1	s_5	s_2	1	1	0	1

(a) Zeichnen Sie das Zustandsdiagramm des Schaltwerks.

(b) Um das Schaltwerk zu realisieren, braucht man jetzt eine Codierung für die Zustände. Wir betrachten zwei von vielen Möglichkeiten:

s_i	Codierung 1 $(z_2z_1z_0)$	Codierung 2 $(z_2z_1z_0)$
s_0	(100)	(001)
s_1	(000)	(110)
s_2	(011)	(000)
s_3	(001)	(101)
s_4	(010)	(100)
s_5	(101)	(010)

Bestimmen Sie für beide Codierungen die Funktionen des Zustandsüberführungsschaltnetzes (das δ-Schaltnetz). Beachten Sie dabei die Möglichkeit von Don't-Cares. Die Tabellen und KV-Diagramme sollten mit abgegeben werden.

(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 Codierung der Zustände erlaubt. Erläutern Sie ihre Vorgehensweise.

(d) Vielfach macht man sich heutzutage keine große Mühe, weil mehr als genug Chipfläche zur Verfügung steht. Man wählt dann einfach eine sog. One-Hot-Codierung, d.h. man wählt im Prinzip für die Zuständscodierung so viele Bits, wie man Zustände hat und weist dann jedem Zustand genau eins dieser Bits zu. Im konkreten Fall würde man also z.B. folgende Codierung andenken:

s_i	$(z_5z_4z_3z_2z_1z_0)$
s_0	(000001)
s_1	(000010)
s_2	(000100)
s_3	(001000)
s_4	(010000)
s_5	(100000)

Wie würden bei dieser Codierung die Funktionen des δ-Schaltnetzes und des λ-Schaltnetzes aussehen? Dazu braucht man keine KV-Disgramme, sondern die Funktionen lassen sich eigentlich sofort hinschreiben.

(e) Oben wurde ganz bewusst die Formulierungen "im Prinzip" und "andenken" verwendet. Die angegebene naive One-Hot-Codierung hat nämlich einen schwer wiegenden Mangel. Welchen? Haben die drei anderen Codierungen, die wir betrachtet haben, diesen Mangel ebenfalls?

Aufgabe 10.2 (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.

Als vierte Alternative sei auf die von Andreas Mäder angebotene virtuelle Maschine hingewiesen.

Für einen ersten Test tippen Sie bitte das folgenden Programm ab oder laden Sie sich die Datei `aufg10_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.

```
/* aufg10_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 aufg10_2.c                     -> aufg10_2.s
 * C -> Objektcode:  gcc -O2 -c aufg10_2.c                     -> aufg10_2.o
 * C -> Programm:    gcc -O2 -o aufg10_2.exe aufg10_2.c        -> aufg10_2.exe
 * Disassembler:     objdump -d aufg10_2.o
 * Ausführen:        aufg10_2.exe
 *
 * 32bit Code auf 64bit System: gcc -m32 ...
 */

#include <stdio.h>

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

  printf( "Meine Matrikelnummer lautet %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) sollten Sie die `gcc`-Compileroption `-m32` verwenden, 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.
Wenn Sie sowohl 64bit Code als auch 32bit Code einschicken und kurz auf die Unterschiede eingehen, gibt es als Weihnachtsgeschenk 10 Punkte extra!

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

Hinweis: Der erzeugte Programmcode (`aufg10_2.exe`) muss nicht mit abgegeben werden. Verschiedene Mailserver halten Mails mit angehängten ausführbaren Programmen wegen eventuell enthaltener Viren automatisch zurück.

Aufgabe 10.3 (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 40`

(b) `LOAD DIRECT 70`

(c) `LOAD INDIRECT 60`

(d) `LOAD DIRECT 80`

(e) `LOAD INDIRECT 30`

Aufgabe 10.4 (Punkte 4·8 + 8)

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

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

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?