PHILIPPS-UNIVERSITÄT MARBURG
Fachbereich Mathematik und Informatik
Prof. Dr. R. Loogen, Dipl.-Inform. J. Beringer

D-35032 Marburg
Hans Meerwein Straße
Lahnberge
18. Januar 2005

Übungen zur „Technischen Informatik I", WS 2004/05

Nr. 12 (letztes Blatt), Abgabe: Dienstag, 25. Januar vor der Vorlesung

---

Die **Klausur** findet am Donnerstag, dem 3. Februar 2005, von 13.15 Uhr bis 15.15 Uhr im Audimax des Hörsaalgebäudes, Biegenstraße 14, statt.
Hilfsmittel sind nicht erlaubt. Mitzubringen ist lediglich Schreibzeug.
Klausureinsicht und -rückgabe:
Mittwoch, 9. Februar 2005, 10.00 - 11.00 Uhr, HG 5 (letzter Vorlesungstermin)

A. Hausaufgaben

61. Interaktives Assemblerprogramm — 4 Punkte

Schreiben Sie ein Assemblerprogramm, das wiederholt ganze Zahlen einliest und aufaddiert, bis eine Null eingegeben wird. Dann soll die aktuelle Summe ausgegeben werden und das Programm terminieren. Testen Sie Ihr Programm im SPIM-Simulator.

62. Konvertierungsprogramm — 5 Punkte

Schreiben Sie ein MIPS-Unterprogramm `itoa` zur Konvertierung eines Integers, der als vorzeichenlose 32-Bit-Binärzahl im Register `$a0` gegeben ist, in einen nullterminierten ASCII-String, der die Zahl dezimal darstellt und dessen Adresse in Register `$a1` übergeben wird. Im Register `$v0` soll die Länge des ASCII-Strings zurückgegeben werden. Das Nullterminierungszeichen soll dabei nicht gezählt werden.

63. Sprungkaskaden — 3 Punkte

Bei der sequentiellen Auswertung eines Booleschen Ausdrucks kann das Endergebnis manchmal schon ohne die Auswertung aller Argumente feststehen. Zum Beispiel folgt in dem Ausdruck `a or b` bereits nach der Auswertung von `a` zu `true`, dass der gesamte Ausdruck den Wert `true` hat. Der Ausdruck `b` braucht in diesem Fall nicht betrachtet zu werden. Man spricht hier von einer *nicht-strikten* Auswertung, da nicht alle Argumente des Booleschen Operators zwingend ausgewertet werden, um das Ergebnis zu ermitteln. Bedingte Sprünge der Form

`if` <Boolescher Ausdruck> `then goto MARKE`

können demnach ohne die Auswertung Boolescher Operatoren allein durch eine Kaskade von Sprüngen realisiert werden.

Stellen Sie auf diese Weise die folgenden Konstrukte in Assembler dar. Die Werte `a` bis `d` seien in den Registern `$a0` bis `$a3` gegeben. Der Wert `true` wird durch eine von Null verschiedene Binärzahl dargestellt, der Wert `false` durch Null.

(a) `if a AND b then goto MARKE`
(b) `if a OR b then goto MARKE`
(c) `if (a XOR b) AND (c XOR b) then goto MARKE`
(d) `if (a OR b) AND (NOT (c XOR d))) then goto MARKE`

B. Mündliche Aufgaben

64. Pseudoinstruktionen

Geben Sie zur Realisierung der folgenden MIPS-Pseudoinstruktionen Sequenzen elementarer MIPS-Instruktionen an. Sie können das Register $at als Hilfsregister verwenden. `big` stehe für eine 32-Bit Konstante und `small` für eine 16-Bit Konstante.

	Instruktion	Bedeutung
(a)	`clear reg`	`reg <- 0`
(b)	`beq reg, small, label`	if `reg = small` goto `label`
(c)	`beq reg, big, label`	if `reg = big` goto `label`
(d)	`lw reg, big(reg')`	`reg <- Memory[big+reg']`

65. Schleifenrealisierung

In imperativen Programmiersprachen existieren Verzweigungskommandos und Schleifenkonstrukte, die in Assembler nicht verfügbar sind. Im folgenden sollen Sie solche Konstrukte in Assemblercodestücke umsetzen, wobei der Bedingungswert im Argumentregister `$v0` gegeben sei. Beispielsweise wird die Alternative

`if` <Bedingung> `then` <Anweisung>

dargestellt durch

```
if     : <Code fuer Bedingungsauswertung mit Ergebnis in $v0>
         beq $v0, $zero, weiter   # Teste Bedingung, 0 = false
then   : <Code fuer Anweisung>
weiter :
```

Setzen Sie die folgenden Konstrukte entsprechend um.

(a) `if` <Bedingung> `then` <Anweisung1> `else` <Anweisung2>

(b) `while` <Bedingung> `do` <Anweisung>

(c) `repeat` <Anweisung> `until` <Bedingung>

(d) `for index :=` <Startwert> `to` <Endwert> `do` <Anweisung>

66. Fließbandverarbeitung

Identifizieren Sie alle Daten-Abhängigkeiten in der nebenstehenden Codesequenz. Welche Daten-Hazards können durch Forwarding aufgelöst werden?

```
add r2, r5, r4
add r4, r2, r5
sw  r5, 100(r2)
add r3, r2, r4
```

67. Hazards

Der folgende Code beinhaltet einen Read-after-Write Hazard, der durch Forwarding aufgelöst werden kann:

```
add r2, r3, r3
add r5, r2, r6
```

Betrachten Sie nun die folgende ähnliche Situation, bei der ein Lesezugriff auf den Speicher nach einem Schreibzugriff auf den Speicher stattfindet:

```
sw r7, 100(r2)
lw r8, 100(r2)
```

Beschreiben Sie kurz, wie sich diese Situation von der vorigen, in der nur Register zum Einsatz kommen, unterscheidet und wie das potentielle Read-after-Write Problem hier gelöst werden kann.