VU Grundlagen der Programmkonstruktion — Übungsteil

Organisatorisches

Wie sollen Ihre Lösungen aussehen

Beachten Sie, dass nicht die konkrete Lösung im Vordergrund steht, sondern der Weg, wie Sie zu dieser gelangen. Geben Sie daher bei Aufgaben immer auch den Rechenweg an. Bei jeder Aufgabe ist ein Beispiel angeführt, wie dieser aussehen soll. Ohne Angabe des Rechenwegs erhalten Sie für die Aufgabe keine Punkte.

Geben Sie ihre Lösungen im TUWEL als (bevorzugt) Text-Dokument (Erweiterung `.txt`) oder als PDF-Dokument (Erweiterung `.pdf`) ab.

Gruppenarbeiten

Sie können (und wir empfehlen es ausdrücklich) die Aufgaben in Gruppen lösen (zu 2-6 Personen). Die Zusammenstellung und Organisation der Gruppen ist dabei Ihnen überlassen. Die Gruppen sollen allerdings für alle Übungseinheiten identisch bleiben und Mehrfachmitgliedschaften sollen ebenfalls nicht vorkommen.

Wenn Sie die Aufgabe in einer Gruppe gelöst haben, so soll EIN Gruppenmitglied die Aufgabe in TUWEL abgeben. Die Namen MIT Matrikelnummer aller Gruppenmitglieder muss im abgegebenden Dokument am Anfang ihres Dokuments vermerkt sein.

Abgabe Ihrer Lösungen

Die Abgabe Ihrer Lösungen erfolgt elektronisch via TUWEL (TU Wien E-Learning Center). Sie finden einen Link dorthin am Beginn der LVA-Seite im TISS ("Zum TUWEL Online-Kurs").

Die Abgabe muss via TUWEL bis spätestens 20. Oktober 2011, 12:00, erfolgen. Nachträgliche Abgaben können ausnahmslos nicht berücksichtigt werden.

Fragen

Bei Fragen zu TUWEL wenden Sie sich bitte zunächst an das TUWEL-Hilfe-System und Ihre Kollegen. Sollte in der Angabe etwas unklar sein, sehen Sie bitte im TUWEL nach, ob dort bereits ergänzende Anmerkungen zum Übungsteil stehen. Wichtige Ankündigungen zum Übungsteil werden auch in der Vorlesung und online bekannt gegeben.

1. Übungsaufgabe

1 Binär- und Hexadezimalzahlen

1. Ermitteln Sie Ihre Matrikelnummer und die von allen ihren Gruppenmitgliedern in Hexadezimal- und Binärdarstellung.
2. Trennen Sie die 16 *Least Significant Bits* ab und berechnen Sie Bit 15...8 `XOR` Bit 7...0. (z.B. wenn die 16 LSB 00101100.00011001 sind, berechnen Sie 00101100 `XOR` 00011001).
3. Wandeln Sie das Ergebnis in Dezimaldarstellung und Hexadezimaldarstellung um.

Geben Sie bei jedem Schritt den Rechenweg an, wie Sie zu Ihrer Lösung gelangt sind.

Beispiel

Beispielzahl: 728. Das Umwandeln kann tabellarisch erfolgen:

Zahl	Dividiert durch 2	Rest	
728	364	0	Bit 0 (LSB)
364	182	0	
182	91	0	
91	45	1	
45	22	1	
22	11	0	
11	5	1	
5	2	1	
2	1	0	
1	0	1	Bit 9 (MSB)

Die Binärdarstellung lautet somit 10.1101.1000b (Im Folgenden werden Binärzahlen von einem “b”, Dezimalzahlen von einem “d” und Hexadezimalzahlen von einem “h” gefolgt.

Die Umwandlung in Hexadezimalzahlen kann direkt aus der Binärdarstellung abgelesen werden: Je 4 Bits repräsentieren eine Ziffer, z.B., entspricht 0010b der Ziffer 2h, 1000b der Ziffer 8h und 1111b der Ziffer Fh.

Die XOR-Verknüpfung von 1101b und 1000b ergibt 0101b. Per Hornerschema kann diese in eine Dezimalzahl umgewandelt werden: $(((0b * 2d) + 1b) * 2d + 0b) * 2d + 1b = 5d$.

2 Assembler

Assemblersprache ist für den Computer lesbarer Maschinencode, der in eine für den Menschen besser verständliche Form gebracht wurde. Das folgende Beispiel

ist solcher (vereinfachter) Assembler-Code (für x86-Prozessoren; die Zahl davor repräsentiert die Speicheradresse, an der der Befehl liegt):

```
1: mov dx 0
2: cmp ax 0
3: je 7
4: add dx ax
5: dec ax
6: jmp 2
7:
```

Der Code benutzt den Befehlszeiger (Program Counter, im Folgenden PC), zwei Register `ax` und `dx`, eine boolsche Variable (ein *Flag*) `zero` und die folgenden Befehle:

`add` ***Register1 Register2*** Berechnet *Register1* + *Register2* und speichert das Ergebnis in *Register1* (äquivalent zu "*Register1* `+=` *Register2*").

`dec` ***Register*** Berechnet *Register* - 1 und speichert das Ergebnis in *Register* (äquivalent zu "*Register*`--`")

`cmp` ***Register Zahl*** Setzt `zero` auf `true`, wenn *Register* gleich *Zahl* ist, ansonsten auf `false` (äquivalent zu "`zero = (` *Register* `==` *Zahl*`)`")

`mov` ***Register Zahl*** Setzt den Wert von *Register* auf *Zahl* (äquivalent zu "*Register* `=` *Zahl*")

`je` ***Adresse*** Bedingter Sprung: Wenn `zero` auf `true` gesetzt ist, springe zu (= setze PC auf) *Adresse*, ansonsten gehe zum nächsten Befehl.

`jmp` ***Adresse*** Unbedingter Sprung: Springe zu (= setze PC auf) *Adresse*.

Nachdem ein Befehl ausgeführt wurde, wird PC um 1 erhöht und somit der nächste Befehl aufgeführt. Eine Ausnahme sind hier die Sprünge `je` (falls `zero` auf `true` gesetzt ist) und `jmp`, die PC auf eine neue Adresse setzen. D.h., wenn PC den Wert 3 hat, so wird der Befehl `je 7` ausgeführt. Sollte das `zero`-Flag gesetzt sein, so enthält danach der PC den Wert 7, ansonsten 4.

Lassen Sie das Programm-Fragment mental laufen und erstellen Sie dabei eine Tabelle, in der Sie den Zustand des PC, der Register, des `zero`-Flags (sofern es von Bedeutung ist) und des nächsten Befehls angeben, bis Sie die Adresse 7 erreichen. Beginnen Sie mit dem Wert 3 in `ax` (die anderen Werte sind am Anfang irrelevant).

PC	`ax`	`dx`	`zero`	Befehl
1	3	?	?	`mov dx 0`
2	3	0	?	`cmp ax 0`
⋮	⋮	⋮	⋮	⋮

Beispiel

Für folgendes Code-Fragment würde Ihre Lösung folgendermaßen aussehen (das Register `dx` wird nicht angeführt, weil es nicht benötigt wird):

```
1: mov ax 2
2: dec ax
3: cmp ax 1
4: je 2
5:
```

PC	`ax`	`zero`	Befehl
1	?	?	`mov ax 2`
2	2	?	`dec ax`
3	1	?	`cmp ax 1`
4	1	`true`	`je 2`
2	1	?	`dec ax`
3	0	?	`cmp ax 1`
4	0	`false`	`je 2`
5	0	?	DONE

3 Grammatiken in EBNF

Gegeben ist folgende Grammatik (Teile in Anführungszeichen repräsentieren Schlüsselwörter bzw. Terminalsymbole):

```
Ausdruck       = Wert { "+" Wert }
               | Wert { "*" Wert }

Wert           = Zahl
               | "(" Ausdruck ")"

Zahl           = [ "-" ] ZifferOhneNull { Ziffer }

Ziffer         = "0" | ZifferOhneNull

ZifferOhneNull = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Werden die folgenden Wörter als `Ausdruck` von der Grammatik akzeptiert oder nicht? Geben Sie jeweils eine (kurze) Begründung an, wie Sie zur Lösung gekommen sind.

- `-0`
- `0251`
- `-12`

- `A + -12`
- `-1 + -1`
- `74 - 99`
- `957 + 123 * -32`
- `12 + ( 642 * 984 * 0 ) + 65`

Beispiel

Das Wort `23 + B` ist kein `Ausdruck`, da `B` kein `Wert` ist (es ist weder eine `Zahl`, noch ein `Ausdruck` in Klammern).

Alternative Erklärung: Das Zeichen `B` kommt in der Sprache nicht vor.

4 Lambda-Ausdrücke

Reduzieren Sie folgenden Lambda-Ausdruck schrittweise, bis Sie zu einer Normalform gelangen. Welche Reduktionsregel wenden Sie an?

$$(\lambda p.(\lambda a.(\lambda b.((p\ b)\ a)))) (\lambda x.(\lambda y.x))$$

Anmerkung: Anstelle des Buchstaben Lambda λ können Sie einen Backslash `\` verwenden. Der Lambda-Ausdruck $\lambda x.(x\ x)$ ist gleich zu `\x.(x x)` und der zu reduzierende Ausdruck ist `(\p. (\a. (\b. ((p b) a)))) (\x.(\y.x))`.

Beispiel

Für den Ausdruck $(\lambda x.(x\ x))\ \lambda y.y$

$(\lambda x.(x\ x))\lambda y.y$ kann mit der β-Regel reduziert werden:

$$(\lambda x.(x\ x))\lambda y.y \rightarrow_\beta [\lambda y.y/x](x\ x)$$

Die weitere Rechnung ergibt $[\lambda y.y/x](x\ x) = ([\lambda y.y/x]x\ [\lambda y.y/x]x) = ((\lambda y.y)\ (\lambda y.y))$.

Auch hier können wir die β-Regel anwenden:

$$((\lambda y.y)\ (\lambda y.y)) \rightarrow_\beta [\lambda y.y\ /\ y]y = \lambda y.y$$

Der letzte Ausdruck kann nicht weiter reduziert werden und ist somit in Normalform.