Aufgabe 21

a) Von der Prozedur werden die Register $t0 bis $t5 verwendet. Inklusive der Rücksprungadresse wird der Stackpointer beim Prozeduraufruf um (6+1)*4=28 Byte verringert (da der Stack nach unten wächst).
b) Diesmal werden (32+1)*4=132 Byte benötigt. Sollte es jedoch möglich sein, die Rücksprungadresse in $ra unterzubringen, so sind nur 32*4=128 Byte notwendig.

Aufgabe 22

a) Der in der Vorlesung vorgestellte Rechner hat die Eigenschaft, dass alle arithmetischen Operationen nur für Register bzw. Immediates definiert sind. Der Ein-Register-Rechner müsste dieses eine Register ständig in den Hauptspeicher sichern, um die Funktionalität des 32-Register-Rechners zu wahren. Dies bedingt ebenfalls enorm viele Bustransfers, die zeitaufwändig sind.
b) Der ursprüngliche Inhalt des Destination-Register ginge ständig verloren, da er nach der Operation schließlich das Rechenergebnis enthält. Oft wird jedoch der ursprüngliche Inhalt mehrfach benötigt, so dass zusätzliche Kopiervorgänge zwischen den Registern notwendig wären. Als Vorteil ergibt sich jedoch, dass die Bit-Kodierung der Befehle 5 Bit einspart, die z.B. für etwaige Spezial-Befehle verwendet werden könnten.

Aufgabe 23

a) Es theoretisch möglich, `beq` durch `bne` zu ersetzen, allerdings müßten dann entweder die Sprungbedingungen negiert oder die Sprungziele verändert werden (d.h. die Befehle, zu denen `beq` springen würde, müssen direkt auf `bne` folgen, während die früher auf `beq` folgenden Befehle jetzt mit `bne` angesprungen werden). Die Negation der Sprungbedingung ist nicht immer möglich bzw. enorm aufwändig, die Vertauschung der Folge-Befehle könnte durch entsprechende Compiler automatisiert werden.
   Eventuell verkomplizieren sich damit Mehrfachalternativen, die viele bedingte Sprünge hintereinander ausführen. Dort könnte unnötig viele zusätzliche Sprünge notwendig werden.
b) Der Befehl `branch` ist ein unbedingter Sprung. Mit `bne` könnte man ihn nachbilden, indem man zwei bedinge Sprünge mit jeweils inversen Bedingen, aber gleichem Sprungziel verwendet. Gemäß den Gesetzen der binären Logik ist davon eine Bedingung immer wahr, so dass das Sprungziel auf jeden Fall erreicht wird. Für `branch` sind demzufolge immer 2 `bne`'s notwendig, so dass sich der Code aufbläht.
c) Dies ist nicht möglich, da `beq` und `bne` bedingte Sprünge sind, `branch` dagegen einen unbedingten Sprungbefehl darstellt. Die fehlende Bedingungseigenschaft von branch erlaubt damit nicht die Konstruktion von Alternativen.
   Es ist lediglich vorstellbar, Sprungtabellen zu verwenden (wie im case-of-Beispiel aus der Vorlesung), ein Register mit dem entsprechenden Sprungziel zu laden und dann `branch $r` ($r sei ein beliebiges Register) auszuführen.

Aufgabe 24

a) Mit der Bedingung, nur `if`s ohne `else` zu verwenden, ergibt sich:

```
if (k==0)
   f=i+j;
if (k==1)
   f=g+h;
if (k==2)
   f=g-h;
if (k==3)
   f=i-j;
```

b) Das dazugehörige Assembler-Programm ist:

```
# Aufgabe 24
#
# von: Stephan Brumme
#
# zuletzt geaendert: 17.Mai 2000
```

```
        .text 0x00400000
        .globl main

main:

# in $t0 lade ich die jeweilige Zahl, mit der verglichen wird
# ist i(=$s5) != $t0, dann wird zum nächsten Vergleich gesprungen (bne-Befehl)
# ansonsten wird addiert, subtrahiert etc. und das Programm verlassen (j exit)

        li      $t0,0
        bne     $t0,$s5,not0
        add     $s0,$s3,$s4
        j       exit
not0:
        li      $t0,1
        bne     $t0,$s5,not0or1
        add     $s0,$s1,$s2
        j       exit
not0or1:
        li      $t0,2
        bne     $t0,$s5,not0or1or2
        sub     $s0,$s1,$s2
        j       exit
not0or1or2:
        li      $t0,3
        bne     $t0,$s5,exit
        sub     $s0,$s3,$s4

exit:
        li      $v0,10                # Programm beenden
        syscall
```