

Prof. L. Thiele

## Technische Informatik 1 - HS 2011

## Übung 2

Datum: 20.10.2011

## 1 Assembler

## 1.1 Wurzelverfahren nach Heron

Das babylonische Verfahren zum Wurzelziehen, das von Heron überliefert wurde, berechnet die Quadratwurzel  $x$  der Zahl  $a$  durch iterative Anwendung der Rechenvorschrift

$$x_{n+1} = \frac{1}{2} \left( x_n + \frac{a}{x_n} \right)$$

mit  $x_0 = a$ . Gegeben sei folgender Assemblercode zur Berechnung der Quadratwurzel von 2 nach dem Heron-Verfahren (10 Iterationsschritte) mit zwei Implementierungsvarianten für die Subroutine `heron`.

Anmerkungen zum Assemblercode:

- Bei den Registern  $f0$ ,  $f2$ ,  $f4$ ,  $f12$  und  $f20$  handelt es sich um Register für Fließkommazahlen.  $f12$  ist das Argument der Subroutine,  $f0$  enthält den Rückgabewert der Subroutine.
- Spezielle Register der MIPS CPU  
 PC: Program Counter. Zeigt auf die Instruktion, welche gerade ausgeführt wird. Die nächste Instruktion befindet sich an der Stelle `Reg[PC] + 4` (Alle Instruktionen sind 32 Bit lang).  
 RA: Return Address. Nach der Ausführung von `JAL`, enthält das Register RA die Adresse der Folge-Instruktion von `JAL`, also `Reg[PC] + 4`. Nach der Ausführung von `JR` in der aufgerufenen Subroutine, wird der Inhalt des Registers RA wieder in das PC-Register geladen. Dies ermöglicht den Rücksprung zur Folge-Instruktion von `JAL`.  
 FP: Frame Pointer. Zeigt auf das erste Argument einer Subroutine.  
 SP: Stack Pointer. Zeigt auf das letzte besetzte Wort auf dem Stack.
- Der Assemblercode ist hinsichtlich Fließkommaoperationen und Verwaltung des Stacks vereinfacht dargestellt. Die Verwaltung des Frames, sowie das Speichern und Rückspeichern von `fp` sind durch `###` ersetzt.

main:

```
###
li    $f12, 2.0e0      # $f12 := 2
li    $a1, 10          # $a1 := 10
jal   heron           # Jump and Link to heron ($ra := $pc + 4)
###
```

## Implementierung 1

```
heron:
    ###
    move    $f2, $f12          # $f2 := $f12
    move    $t1, $0            # $t1 := 0 (Register $0 ist immer Null)
    ble     $a1, $0, $Label3   # if ($a1 <= $0) then goto $Label3
    li      $f4, 5.0e-1        # $f4 := 0.5

$Label5:
    div     $f0, $f12, $f2     # $f0 := $f12 / $f2
    add     $f0, $f2, $f0      # $f0 := $f0 + $f2
    mul     $f0, $f0, $f4      # $f0 := f0 * $f4
    ???
    slt     $t0, $t1, $a1      # ==> Hier fehlt eine Zeile
                                # if ($t1 < $a1) then $t0 := 1
                                #     else $t0 := 0
    move    $f2, $f0           # $f2 := $f0
    bne     $t0, $0, $Label5   # if ($t0 <> $0) then goto $Label5

$Label3:
    move    $f0, $f2           # $f0 := $f2
    ###
    jr      $ra                # Jump and return ($pc := $ra)
```

## Implementierung 2

```
heron:
    ###
    sw      $f20, 24($sp)      # Frameverwaltung inkl. Aenderung des SP
                                # Memory[$sp+24] := $f20
    move    $f20, $f12         # $f20 := $f12
    sw      $ra, 16($sp)       # Memory[$sp+16] := $ra
    bgt     $a1, $0, $Label2   # if ($a1 > $0) then goto $Label2
    move    $f0, $f20          # $f0 := $f20
    j       $Label4            # jump to $Label4
$Label2:
    move    $f12, $f20         # $f12 := $f20
    addi    $a1, $a1, -1       # $a1 := $a1 - 1
    jal     heron              # Jump and Link to heron
    div     $f2, $f20, $f0     # $f2 := $f20 / $f0
    add     $f2, $f2, $f0      # $f2 := $f2 + $f0
    li      $f0, 5.0e-1        # $f0 := 0.5
    mul     $f2, $f2, $f0      # $f2 := f2 * $f0
    move    $f0, $f2

$Label4:
    lw      $ra, 16($sp)       # $ra:=Memory[$sp+16]
    lw      $f20, 24($sp)      # $f20:=Memory[$sp+24]
    ###
    jr      $ra                # Jump and return, ($pc := $ra)
```

- (a) Implementierung 1 ist unvollständig. Was fehlt? Ergänzen Sie das Assemblerprogramm, indem Sie an der mit ??? markierten Stelle eine Zeile einfügen.
- (b) Geben Sie zu Implementierung 2 ein möglichst kompaktes Programm in C oder Java an, das den wesentlichen Programmablauf in Implementierung 2 widerspiegelt.
- (c) Wo liegt der prinzipielle Unterschied zwischen der (vervollständigten) Implementierung 1 und Implementierung 2? Welche von beiden ist schneller?

## 1.2 Fibonacci-Zahlen

Die folgende rekursive Rechenvorschrift ermöglicht Ihnen, die "Fibonacci-Zahlen" zu berechnen:

$$Fib(n) = Fib(n - 1) + Fib(n - 2)$$

$$Fib(1) = Fib(2) = 1$$

Diese soll nun in Assembler implementiert werden. Um Ihnen die Arbeit etwas zu vereinfachen, ist der erste Teil des Assembler-Programms vorgegeben:

```
.data
in_string: .asciiz "Bitte geben Sie eine ganze Zahl ein:\n"
out_string: .asciiz "Die Fibonacci-Zahl lautet:\n"

.text
__start:
    puts in_string      # Ausgabe des Strings
    geti $s0           # Einlesen der Eingabe in s0

    addi $sp, $sp, -4  # Erniedrigen des Stack Pointer
    sw   $s0, 0($sp)  # Das Argument fuer die
                    #   Fib Subroutine wird auf den
                    #   Stack gelegt.

    jal  Fib          # Sprung zur Fib Subroutine
    lw   $s1, 0($sp)  # Hole das Resultat vom Stack
    addi $sp, $sp, 4  # Erhoehen des Stack Pointer
    puts out_string   # Ausgabe des Strings
    puti $s1          # Ausgabe der Fibonacci-Zahl
done
```

Es ist empfehlenswert die Subroutine Fib in drei Teile aufzuspalten:

- (a) Vorbereitungsphase: In diesem Teil werden die notwendigen Vorbereitungen getroffen, damit der Stack sauber aufgebaut wird. Hier sollte auch die Rekursionsbedingung stehen: `if (n > 2) then rekursion else abbauphase`
- (b) Rekursion: Hier erfolgt der eigentliche Aufruf von `Fib(n-1)` und `Fib(n-2)`. Hier steht auch die entsprechende Rekursionsgleichung.
- (c) Abbauphase: Hier wird einerseits der Stack wieder abgebaut und andererseits das Resultat auf den Stack gelegt, damit es in der Phase 2 abgeholt werden kann.

```

# Subroutine Fib
# Eingabe: Integer n, auf dem Stack
# Ausgabe: Fib(n), ebenfalls auf dem Stack

Fib:
# Vorbereitungsphase:
...
sw    $ra, 4($sp)           # Da wir mit mehrfach verschachtelten
                           # Subroutinen arbeiten, muessen wir
                           # die Return Adresse ra auf dem
                           # Stack sichern.

...
bgt   ... , ..., rekursion # Rekursionsbedingung und Sprung zur
                           # Rekursionsphase

...
j     abbauphase           # Zur Abbauphase springen

rekursion:
...
jal   Fib                 # Fib(n-1)
...
jal   Fib                 # Fib(n-2)
...
abbauphase:
...
jr    $ra                 # Return Adresse vom Stack holen
                           # Ruecksprung

```

Tipps: Gehen Sie am besten wie folgt vor:

- (a) Schreiben Sie die Routine kurz in C oder Java auf und bringen Sie ihren Code mit dem gegebenen Assembler-Skelett in Beziehung.
- (b) Überlegen Sie, welche Register sie benötigen, und welche Register gesichert werden müssen. Zeichnen Sie auf, wie Ihr Stack aussehen muss. Vergessen Sie nicht, Stack-Pointer und Frame-Pointer nachzuführen und beachten Sie die Verwaltung der Rücksprungadressen.
- (c) Implementieren Sie die Frameverwaltung. Kontrollieren Sie, dass alle gesicherten Variablen am Ende der Funktion wiederhergestellt sind. Achten Sie insbesondere darauf, dass der Stack-Pointer wieder den gleichen Wert hat.
- (d) Implementieren Sie schliesslich den Basisfall und die rekursiven Aufrufe.