Aufgabe 1: Assemblerprogrammierung Lösungsvorschlag

Gegeben sei das folgende C-Programm und ein entsprechendes MIPS-Assemblerprogramm. Der implementierte Algorithmus testet, ob die Ganzzahl n eine Primzahl ist.

```
/* C-Code */
/* ------ */
int factorNotInRange(int k, int n) {
  if (k >= n)
       return 1;                                        /*returntrue*/
  else if ((n % k) == 0)
       return 0;                                        /* return false */
  else
       return factorNotInRange(k+1, n);
}

int main() {
  int n=5;                                              /* Beispiel n=5 */
  return ((n > 1) && factorNotInRange(2, n));
}
```

```
#MIPS-Assembler
#--------------
.text
.globl main
main:
       li      $a1, 5            #$a1:=5, Beispiel n=5
       li      $a0, 2
       li      $s2, 1
       sgt     $s1, $a1, $s2     #if($a1>$s2)then $s1:=1 else $s1:=0
       beq     $s1, $zero, end
       jal     FactorNotInRange
end:
       and     $s7, $s1, $v0     #$s7 enthält das Resultat des Tests
       li      $v0, 10           #diese beiden Zeilen stoppen...
       syscall                   #... das Ausführen des Codes im Simulator
FactorNotInRange:
                                 #hier ergänzen (mehrere Zeilen)
       move    $s0, $a0          #$s0:=$a0
       move    $s1, $a1

       sge     $v0, $s0, $s1     #if($s0>=$s1) then $v0:=1 else $v0:=0
       bne     $v0, $zero, exit

       rem     $t1, $s1, $s0     #$t1:=Rest($s1/$s0)
       sne     $v0, $t1, $zero   #if($t1=!=$zero) then $v0:=1 else $v0:=0
       beq     $t1, $zero, exit

       addi    $a0, $s0, 1
       move    $a1, $s1
       jal     FactorNotInRange
exit:
                                 #hier ergänzen (mehrere Zeilen)
```

Aufgaben:

a) Der MIPS-Assembler Code enthält einige Lücken. Fügen Sie bei der mit dem Kommentar “hier ergänzen” bezeichneten Stellen die fehlenden Zeilen ein. Halten Sie sich an allfällige Konventionen. Nehmen Sie nur die für eine fehlerfreie Ausführung des Programms notwendigen Ergänzungen vor.

```
1. Lücke
       addiu   $sp, $sp, -12     # 1.5 P.
       sw      $ra, 8($sp)       # 1 P.
       sw      $s0, 4($sp)       # 0.5 P.
       sw      $s1, 0($sp)       # 0.5 P.

2.Lücke

       lw      $s1, 0($sp)       # 0.5 P.
       lw      $s0, 4($sp)       # 0.5 P.
```

```
        lw      $ra, 8($sp)       # 1 P.
        addiu   $sp, $sp, 12      # 1.5 P.
        jr      $ra               # 2 P.
```

$fp Manipulation je nach Konvention (GNU <-> MIPS) nicht notwendig, daher nicht bewertet.

b) *li rt, immediate* ist eine sogenannte Pseudoinstruktion. Diese kann nicht direkt auf dem MIPS Prozessor ausgeführt werden und muss daher beim Assemblieren in einen oder mehrere MIPS-Instruktionen übersetzt werden. Notieren Sie die komplette(n) MIPS-Instruktion(en) für *li $a1, 5*. Benützen Sie möglichst wenig Instruktionen.

```
        ori $a1, $zero, 5
oder
        addi $a1, $zero, 5
oder
        addiu $a1, $zero, 5
```

c) Diese Implementation des Algorithmus ist nicht sehr effizient. Weshalb? Schreiben Sie einen möglichst kurzen C-Code, der diese Ineffizienz vermeidet, ohne den Algorithmus grundsätzlich zu verändern.

Programm ist nicht effizient, da rekursiv (1 P.). Bessere Implementation mit Schleife (1 P.).

Bsp. Code (2 P.)

```
int main() {
        int n,k;
        n=5;
        if (n<=1) {
                return 0;
        } else {
                for (k=2; k<n; k++) {
                        if ((n%k)==0) {
                                return 0;
                        }
                }
        }
return 1;
}
```