## Aufgabe 33

a) Der Pseudobefehl `move $rd,$rs` wird als `addu $rd,$0,$rs` übersetzt. Dabei macht sich SPIM zunutze, dass das Register `$0` immer Null ist. Somit wird das Register `$rd` ersetzt durch `$rd`=0+`$rs`=`$rs`, was einem `move`-Befehl gleichkommt. Der Vorteil dieser Technik ist, dass `move` nicht hardwaremäßig implementiert werden muss, dem Programmierer aber gleichzeitig der Komfort dieses Befehls geboten wird.
Ebenfalls denkbar ist die Ausnutzung der Beziehung a∨0=a, der `move`-Befehl könnte also auch mit or `$rd,$0,$rs` übersetzt werden. Auf ähnliche Art und Weise lassen sich noch diverse andere binäre Verknüpfungen konstruieren, z.B. mit `and`, `xor` etc .

b) In einem allgemeinen Pseudocode lautet der `swap`-Befehl:

```
temp:=a;
a:=b;
b:=temp;
```

Die Umsetzung in Assembler mittels `xor` benötigt weiter dessen Eigenschaft, dass a ⊕ 0 = a:

```
xor $at,$0,$r1
xor $r1,$0,$r2
xor $r2,$0,$at
```

($r1 und $r2 stehen stellvertretend für beliebige Register)
Als Programm kann man diesen Code jedoch nicht in SPIM eingeben, da es nicht zulässig ist, das `$at`-Register zu benutzen, es ist nur dem Assembler vorbehalten, der allerdings selbst dann diese Sequenz generieren könnte. Zur Verdeutlichung des Funktion der `xor`-Befehls die dazugehörige Verknüpfungstabelle, die zeigt, dass für b=0 stets a ⊕ b=a, d.h. a ⊕ 0=a:

| a | b | a ⊕ b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

```
# Aufgabe 33: Pseudobefehle
#
# von: Stephan Brumme
#
# zuletzt geaendert: 07.Juni 2000


        .text
main:

# Aufgabe a
        move    $t0,$t1                 # verschiedene Ersetzungen von move
        addu    $t0,$0,$t1
        or      $t0,$0,$t1

# Aufgabe b
        xor     $t7,$0,$t0              # swap $t0,$t1
        xor     $t0,$0,$t1              # Register $at kann aber nicht direkt benutzt werden
        xor     $t1,$0,$t7              # deshalb verwende ich hier $t7

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

## Aufgabe 34

Der Test einer Zahl auf ihre Primeigenschaft erfolgt, indem ich sie durch alle Zahlen, die größer 1 und kleiner als die Zahl selbst sind, dividiere und den entstehenden Rest überprüfe. Ist er 0, so hat die Zahl noch andere Teiler außer den trivialen und kann keine Primzahl sein.

In Pseudocode sieht das Programm folgendermaßen aus:

```
int test_prime(arg)
{
	int i = arg;		// von arg-1 bis 2 alle Zahlen als potenzielle Teiler testen

	do
	{
		i--;
		if (i==1) return 1;	// keinen nicht-trivialen Primfaktor gefunden => Primzahl
	}
	while (arg%i>0);		// Prüfung, ob Division ohne Rest moeglich

	return 0;			// Primfaktor gefunden => keine Primzahl
}

main()
{
	n = 1;				// von 2 bis 100 alle natuerlichen Zahlen testen
loop:
	n++;
	if (n>100) halt;		// alle Zahlen getestet ?

	if (test_prime(n)==1)		// wenn Primzahl, dann Bildschirmausgabe
		printf(„%d „,n);

	goto loop;
}
```

In SPIM-Assembler dann:

```
# Aufgabe 34: Unterprogramme / Primzahlberechnung
#
# von: Stephan Brumme
#
# zuletzt geaendert: 07.Juni 2000

	.data
leerzeichen: .asciiz " "

	.text
main:
	li	$s0,1		# bei 2 anfangen (die 1 wird im uebernaechsten Befehl erhoeht)
	li	$s1,100		# alle Zahlen bis 100 testen

main_loop:
	addi	$s0,$s0,1	# naechste Zahl
	bgt	$s0,$s1,main_done	# >100 ? ja, dann fertig

	move	$a0,$s0		# zu testende Zahl in $a0 als Argument laden
	jal	test_prime	# auf Primzahl testen
	beqz	$v0,main_loop	# nicht prim, also keine Bildschirmausgabe

	li	$v0,1		# Integer ausgeben
	syscall			# Hinweis: $a0 enthaelt bereits die Zahl !

	li	$v0,4		# Leerzeichen zur optischen Auflockerung ausgeben
	la	$a0,leerzeichen
	syscall

	j	main_loop	# naechster Schleifendurchlauf

main_done:
	li	$v0,10		# Programm beenden
	syscall



# ##############################################################################################
# TEST_PRIME:
# UP, das eine einzelne Zahl testet, ob sie prim ist
#
# Parameter:	$a0 = zu testende Zahl
# Rückgabe:	$v0 = 1 wenn prim, 0 wenn nicht prim

test_prime:
```

```
        li      $v0,1                   # davon ausgehen, dass Zahl prim ist

        move    $t0,$a0                 # bei n beginnen
        li      $t1,1                   # konstante Hilfsvariable,
                                        # fuer Schleifen und Vergleiche

test_prime_loop:
        sub     $t0,$t0,$t1             # sie ist sicher durch sich selbst teilbar, also
                                        # die naechstkleinere Zahl verwenden

        beq     $t0,$t1,test_prime_done # wenn 1 erreicht ist, dann UP beenden

        remu    $t2,$a0,$t0             # Rest der Division $a0/$t0 ermitteln
        bgtz    $t2,test_prime_loop     # wenn Rest existiert, dann ist $t0 kein Primfaktor
                                        # von $a0, also naechste Zahl untersuchen

        li      $v0,0                   # doch Primfaktor gefunden => keine Primzahl
test_prime_done:
        jr      $ra                     # UP beenden
```

Man könnte das Unterprogramm noch erheblich beschleunigen, wenn man nur die Zahlen von 2 bis $\sqrt{n}$ untersucht, ob sie Teiler von *n* sind (d.h. die Teilbarkeit durch 2,3,5 und 7 testet). Ebenso ist es möglich, von Anfang an alle geraden Zahlen nicht zu testen, da sie, von der 2 abgesehen, nie prim sind. All diese Tricks blähen den Code jedoch unnötig auf und wurden deshalb nicht implementiert.

## Aufgabe 35

```
# Aufgabe 35: Unterprogramme / Stack
#
# von: Stephan Brumme
#
# zuletzt geaendert: 07.Juni 2000

        .data

        .text
main:
        li      $a0,42                  # zwei Werte auf den Stack legen
        jal     push
        li      $a0,23
        jal     push

        jal     pop                     # und wieder auslesen
        jal     pop

        li      $v0,10                  # Programm beenden
        syscall


# ##############################################################################################
# POP:
# UP, das eine Zahl vom Stack holt
#
# Parameter:   keine
# Rückgabe:    $v0 = Zahl aus dem Stack
pop:
        add     $sp,$sp,4               # ein belegter Platz, d.h. 4 Byte, weniger im Stack
        lw      $v0,($sp)               # Zahl von den Stackspitze auslesen
        jr      $ra                     # UP beenden


# ##############################################################################################
# PUSH:
# UP, das eine Zahl auf den Stack legt
#
# Parameter:   $a0 = Zahl, die auf den Stack gelegt wird
push:
        sw      $a0,($sp)               # Element dort speichern
        add     $sp,$sp,-4              # Zeiger auf naechste freie Stelle weiterbewegen
        jr      $ra                     # UP beenden
```

## Aufgabe 36

a) Der Befehl `rol rd,rs1,rs2` schiebt `rs1` um `rs2` Bits nach links und fügt die herausgeschobenen Bits rechts wieder ein. Das Ergebnis dieser Operation steht dann in `rd`, wobei `rs1` und `rs2` selbst nicht verändert werden.
Es wird dabei `rs1` nur als „Bit-Ansammlung“ betrachtet und nicht als Zahl, die eventuell im Zweierkomplement steht. Die bei Schiebeoperationen mögliche Interpretation als Multiplikation bzw. Division ist daher nicht möglich. In der nachfolgenden Tabelle wird dies gezeigt:

| rs1 | rs2 | rd |
|---|---|---|
| 0x1248 8421 | 16 | 0x8421 1248 |
| 0x1248 8421 | 8 | 0x4884 2112 |
| 0x1248 8421 | 4 | 0x2488 4211 |
| 0xF0FF FFF0 | 16 | 0xFFF0 F0FF |
| 0xF0FF FFF0 | 8 | 0xFFFF F0F0 |
| 0xF0FF FFF0 | 4 | 0x0FFF FF0F |

Die dezimale Schreibweise ergibt z.B. für die letzte Zeile in Pseudo-Assembler:
`rol –25168256,4 = 268435215`.

Es bleibt noch zu erwähnen, dass auch dieser `rol`-Befehl (ähnlich wie `move`) nur ein Pseudobefehl ist, der durch eine Sequenz von 4 Befehlen ersetzt und damit auf Schiebe- und `or`-Instruktionen zurückgeführt wird.

b) Für `sra rd,rs1,imm` errechnet sich bei den vorgegebenen Werten:

| rs1 | imm | rd |
|---|---|---|
| 0x1248 8421 | 0100 | 0x0124 8842 |
| 0x1248 8421 | 1000 | 0x0012 4884 |
| 0x1248 8421 | 1101 | 0x0000 9244 |
| 0xFFFF FFFC | 0100 | 0xFFFF FFFF |
| 0xFFFF FFFC | 1000 | 0xFFFF FFFF |
| 0xFFFF FFFC | 1101 | 0xFFFF FFFF |

Negative Werte beeinflussen den Befehl dahingehend, dass das Rechtsschieben um n Bits neue n MSBits erzwingt. Um das Zweierkomplement nicht zu verändern, wird jeweils das höchstwertigste Bit repliziert, so dass negative Zahlen auch negativ bleiben. Dies erlaubt die Zweckentfremdung für schnelle Multiplikationen / Divisionen mit Zweierpotenzen.

```
# Aufgabe 36: Schiebe- und Rotationsbefehle
#
# von: Stephan Brumme
#
# zuletzt geaendert: 07.Juni 2000

        .text 0x400000
main:

# Aufgabe a
        li      $s1,0x12488421

        li      $s0,16
        rol     $s2,$s1,$s0

        li      $s0,8
        rol     $s3,$s1,$s0

        li      $s0,4
        rol     $s4,$s1,$s0


        li      $s1,0xF0FFFFF0
```

```
        li      $s0,16
        rol     $s5,$s1,$s0

        li      $s0,8
        rol     $s6,$s1,$s0

        li      $s0,4
        rol     $s7,$s1,$s0


# Aufgabe b

        li      $s1,0x12488421

        li      $s0,4
        sra     $t0,$s1,$s0

        li      $s0,8
        sra     $t1,$s1,$s0

        li      $s0,13
        sra     $t2,$s1,$s0


        li      $s1,-4

        li      $s0,4
        sra     $t3,$s1,$s0

        li      $s0,8
        sra     $t4,$s1,$s0

        li      $s0,-4
        sra     $t5,$s1,$s0


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