

## Rechnerarchitektur

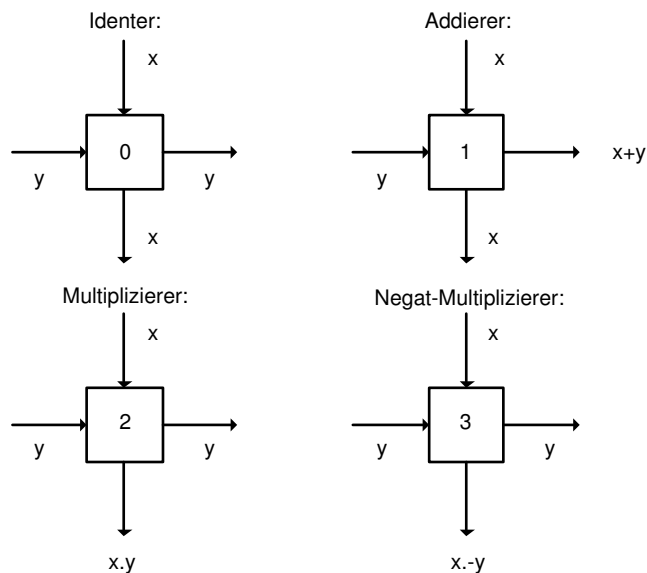
**Abgabetermin:** 09.06.2008, 12:00 Uhr

**Achtung:** Am Mittwoch, dem 18. Juni (vorauss. 14:30 Uhr) findet im Rahmen der Vorlesung "Programmierung und Modellierung" eine Info-Veranstaltung zur Praktikumswahl statt. Informationen unter <http://www.nm.ifl.lmu.de/teaching/Praktika/2008ws/sysprak/>

**Lesen:** Rechnerarchitektur-Skript: Kapitel 13: Einführung in den SPIM Simulator. Programme und weitere Informationen zu SPIM sind auf der Vorlesungshomepage zu finden.

### Aufgabe 18: (H) Programmierbare logische Arrays (2+4+2+2+2+6 Pkt.)

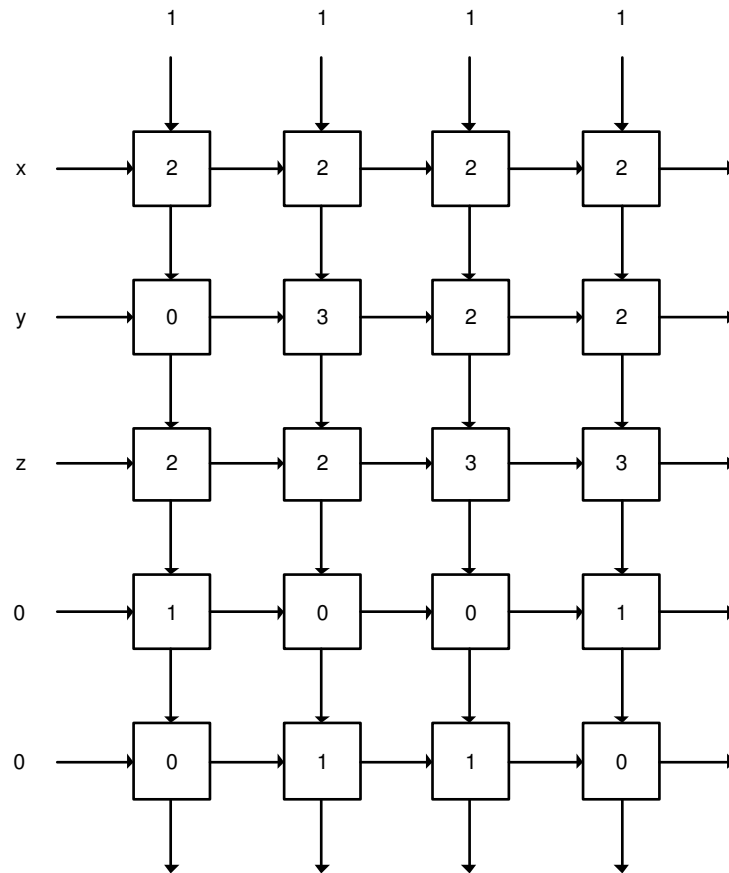
- In der Vorlesung haben Sie das Konzept von programmierbaren logischen Arrays (PLAs) kennen gelernt. Erläutern Sie kurz die grundlegende Idee eines PLAs.
- Intern ist jeder PLA gitterförmig verdrahtet, wobei sich an jedem Kreuzungspunkt von zwei Drähten einer von vier möglichen Bausteinen befindet. Diese Bausteine sind:



Zeichnen Sie das Schaltbild für jeden der vier Bausteine. Verwenden Sie dazu Und-, Oder- und Nicht-Gatter.

- Wie Sie wissen, sind im Prinzip alle Booleschen Funktionen z.B. mit Nand-Gattern realisierbar. Warum werden trotzdem vier verschiedene Bausteine in PLAs eingesetzt?

- d. Erläutern Sie, was es bedeutet, wenn Eingänge
- (i) neutralisiert werden.
  - (ii) gesperrt werden.
- e. Ein normierter PLA besteht aus einer Und-Ebene und aus einer Oder-Ebene. Erklären Sie diese beiden Begriffe kurz. Ausgehend von einem 5-mal-4-PLA: Wie groß werden Und- und Oder-Ebene jeweils, wenn durch den PLA eine dreistellige Boolesche Funktion realisiert werden soll?
- f. Welche Boolesche Funktion berechnet der folgende normierte PLA?



**Aufgabe 19: (H) Test des Simulators**

(1+5+1 Pkt.)

Für diese Aufgabe sollten Sie sich mit dem MIPS-Simulator SPIM vertraut machen. Sie können einen MIPS-Simulator von der Vorlesungshomepage herunterladen, oder XSPIM aus dem CIP-Pool benutzen. Laden Sie bitte das Programm „simple.s“ in den MIPS-Simulator. Wenn Sie XPIM benutzen, sind hier kurz die dafür notwendigen Schritte dargestellt:

- Laden Sie sich das Assemblerprogramm `simple.s` von der Rechnerarchitektur-Homepage herunter und speichern Sie es in Ihrem Home-Verzeichnis ab.
  - Geben Sie auf einem Terminal `xspim` ein, um den Simulator starten.
  - In dem erscheinenden Fenster wählen Sie `load`, geben dann den Dateinamen der Programmdatei ein (`simple.s`) und klicken auf `assembly file`.
  - Nun können Sie das Programm mit `run` ausführen. Dabei sollte eine Konsole erscheinen, über die die Ein- und Ausgabe erfolgt.
- a. Welches Ergebnis liefert das Programm für die Eingabefolge "1, 2, 3, 4, 0"? (D.h. nach Start des Programms erfolgt über die Konsole die Eingabe "1", gefolgt von *Enter*, dann die Eingabe "2", gefolgt von *Enter*, usw.)
- b. Kommentieren Sie die mit Fragezeichen gekennzeichneten Zeilen des Programms sinnvoll. (Geben Sie bei Ihrer Abgabe die Zeilennummer und den zugehörigen Kommentar an.)
- c. Welche mathematische Funktion berechnet das Programm?

**Aufgabe 20: (H) Boolesche Funktionen**

(2+2 Pkt.)

Gegeben sei die boolesche Funktion  $g$ :

$$g(a, b, c, d) = d + \neg d \cdot c \cdot a + \neg d \cdot c \cdot \neg a.$$

- a. Geben Sie die DNF von  $g$  an.
- b. Geben Sie die KNF von  $g$  an.

**Aufgabe 21: (T) Entwurf eines 4-Bit-Addiernetzes**

(5+5+5 Pkt.)

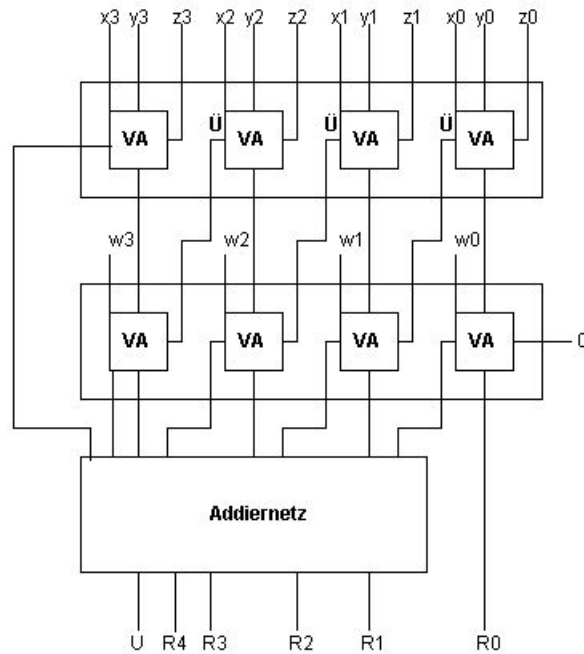
Es soll systematisch ein Addiernetz entworfen werden, das in der Lage ist, zwei 4-stellige Dualzahlen zu addieren. Dazu wird das Problem aufgespaltet, indem man überlegt, wie eine einzelne Stelle addiert wird.

- a. Entwerfen Sie einen Halbaddierer, der in der Lage ist, zwei einstellige Dualzahlen zu addieren.
- b. Entwerfen Sie einen Volladdierer, der in der Lage ist, eine beliebige Stelle zweier  $n$ -stelliger Dualzahlen zu addieren.
- c. Entwerfen Sie nun das Addiernetz, indem Sie Halb- und Volladdierer verwenden.

**Aufgabe 22: (T) Carry-Save-Addiernetz**

(2+2+2+4 Pkt.)

Betrachten Sie die Funktionsweise und den Aufbau des Carry-Save-Addiernetzes:



- Welche Unterschiede bestehen zum Ripple-Carry-Addierer? Welche Vor- und Nachteile bestehen insbesondere bezüglich Zeitverhalten bzw. Schaltungsaufwand?
- Berechnen und vergleichen Sie die Zeit, die die einzelnen Addiernetze zur Addition von vier 8-Bit Zahlen benötigen. Nehmen Sie dazu an, daß die Zeitverzögerung für jedes grundlegende Schaltelement 1 Takt beträgt.

*Hinweis:* Halb- und Volladdierer gelten nicht als grundlegende Schaltelemente.

## SPIM-Programm: simple.s

```

        .data
2  str1:  .ascii  "Geben_Sie_beliebig_viele_Zahlen_zwischen_1_und_99_ein.\n"
        .asciiz  "Eingabe_von_0_beendet_die_Eingabe_und_gibt_das_Ergebnis_aus.\n"
4  askstr: .asciiz  "\n?->"
        errstr: .asciiz  "Sie_dürfen_nur_Zahlen_zwischen_1_und_99_eingeben.\n"
6  ansWSTR: .asciiz  "Das_Ergebnis_lautet:_"
        str2:  .asciiz  "\n\n"
8
        .text
10 main:  li      $s0, 0          # --- ???
        li      $s1, 0          # --- ???
12
        li      $v0, 4
14        la      $a0, str1
        syscall          # PRINT_STR(str1)
16
loop:    li      $v0, 4
18        la      $a0, askstr
        syscall          # PRINT_STR(askstr)
20
        li      $v0, 5
22        syscall          # $v0 := READ_INT;
        li      $t2, 99        # $t2 := 99;
24        bgt     $v0, $t2, error # --- ???
        li      $t2, 0         # $t2 := 0;
26        blt     $v0, $t2, error # --- ???
        beqz    $v0, exit      # --- ???
28        addi    $s1, $s1, 1    # --- ???
        mul     $t2, $v0, $v0   # --- ???
30        mul     $t2, $t2, $s1  # --- ???
        add     $s0, $s0, $t2   # --- ???
32        j       loop          # --- ???

34 error: li      $v0, 4
        la      $a0, errstr
36        syscall          # PRINT_STR(errstr)
        j       loop
38
exit:    li      $v0, 4
40        la      $a0, ansWSTR
        syscall          # PRINT_STR(ansWSTR)
42
        li      $v0, 1
44        move    $a0, $s0
        syscall          # PRINT_INT($s0)
46
        li      $v0, 4
48        la      $a0, str2
        syscall          # PRINT_STR(str2)
50
        li      $v0, 10        # Systemaufrufnr. 10 = EXIT
52        syscall

```

## SPIM Assemblerbefehle

Befehl	Argumente	Wirkung
add	Rd, Rs1, Rs2	$Rd := Rs1 + Rs2$
addu	Rd, Rs1, Rs2	$Rd := Rs1 + Rs2$
addi	Rd, Rs1, Imm	$Rd := Rs1 + Imm$
addiu	Rd, Rs1, Imm	$Rd := Rs1 + Imm$
div	Rd, Rs1, Rs2	$Rd := Rs1 \text{ DIV } Rs2$
rem	Rd, Rs1, Rs2	$Rd := Rs1 \text{ MOD } Rs2$
mul	Rd, Rs1, Rs2	$Rd := Rs1 \times Rs2$
b	label	unbedingter Sprung nach label
j	label	unbedingter Sprung nach label
jal	label	unbed. Sprung nach label, Adresse des nächsten Befehls in \$ra
jr	Rs	unbedingter Sprung an die Adresse in Rs
beq	Rs1, Rs2, label	Sprung, falls $Rs1 = Rs2$
beqz	Rs, label	Sprung, falls $Rs = 0$
bne	Rs1, Rs2, label	Sprung, falls $Rs1 \neq Rs2$
bnez	Rs1, label	Sprung, falls $Rs1 \neq 0$
bge	Rs1, Rs2, label	Sprung, falls $Rs1 \geq Rs2$
bgeu	Rs1, Rs2, label	Sprung, falls $Rs1 \geq Rs2$
bgez	Rs, label	Sprung, falls $Rs \geq 0$
bgt	Rs1, Rs2, label	Sprung, falls $Rs1 > Rs2$
bgtu	Rs1, Rs2, label	Sprung, falls $Rs1 > Rs2$
bgtz	Rs, label	Sprung, falls $Rs > 0$
ble	Rs1, Rs2, label	Sprung, falls $Rs1 \leq Rs2$
bleu	Rs1, Rs2, label	Sprung, falls $Rs1 \leq Rs2$
blez	Rs, label	Sprung, falls $Rs \leq 0$
blt	Rs1, Rs2, label	Sprung, falls $Rs1 < Rs2$
bltu	Rs1, Rs2, label	Sprung, falls $Rs1 < Rs2$
bltz	Rs, label	Sprung, falls $Rs < 0$
syscall		führt Systemfunktion aus
move	Rd, Rs	$Rd := Rs$
la	Rd, label	Adresse des Labels wird in Rd geladen
lb	Rd, Adr	$Rd := \text{MEM}[\text{Adr}]$
lw	Rd, Adr	$Rd := \text{MEM}[\text{Adr}]$
li	Rd, Imm	$Rd := Imm$
sw	Rs, Adr	$\text{MEM}[\text{Adr}] := Rs$

Funktion	Code in \$v0	Funktion	Code in \$v0
print_int	1	read_float	6
print_float	2	read_double	7
print_double	3	read_string	8
print_string	4	sbrk	9
read_int	5	exit	10

## Bemerkung:

Alle arithmetischen Befehle (mul, ...), Sprungbefehle und Vergleichsbefehle sind auch mit einem Imm-Argument (Immediate-Argument) statt Rs2 möglich. Zum Beispiel: `$t0, $t1, 5` statt `li $t5, 5` gefolgt von `mul $t0, $t1, $t5`. Dies funktioniert, da der Assembler dann den Wert zunächst in sein \$at-Register lädt.