Beispielhafte Prüfungsaufgaben zur Vorlesung
Technische Informatik I
Gestellt im Frühjahr 2010

Die beigefügte Lösung ist ein Vorschlag. Für Korrektheit, Vollständigkeit und Verständlichkeit wird keine Verantwortung übernommen.

Aufgabe 4 : Cache (maximal 35 Punkte)

4.1: Cachegrössen (maximal 10 Punkte)

Für einen 32-bit Prozessor soll ein Cache implementiert werden, der 64kByte an Daten fassen soll (1kByte = 2^{10}Byte). Der Cache ist folgendermassen spezifiziert:

- Adressierungsart bei Speicherinstruktionen: Wortadressierung
- Pro Cacheblock 16 Wörter à 4 Bytes
- Von den Statusbits wird nur das Valid-Bit berücksicktigt

Berechnen Sie die effektive Cachegrösse in Bits für:

(a) Rechner A: direkte Abbildung

Lösungsvorschlag:

Zuerst wird die benötigte Anzahl Blöcke berechnet. Es sollen 64kB an Daten gespeichert werden. Ein Wort enthält 4 Bytes. Der Speicher muss also 16kWörter speichern. Pro Block werden 16 Wörter gespeichert, also braucht es 1kBlöcke, was einer Cachegrösse von $2^{10} = 1024$ Blöcken entspricht.

$2^{10} \times (32 \cdot 16 + 32 - 10 - 4 + 1)Bit = 543'744Bit$
Blöcke $\times$ (Datenbits + Adressbits − Index-Bits − Word-Offset + Valid-Bit)

□

(b) Rechner B: teilassoziativ mit 8 Einträgen pro Index

Lösungsvorschlag:

Anzahl Blöcke wie in (a).

$2^{10} \times (32 \cdot 16 + 32 - 7 - 4 + 1)Bit = 546'816Bit$
Blöcke $\times$ (Datenbits + Adressbits − Index-Bits − Word-Offset + Valid-Bit)

□

(c) Rechner C: vollassoziativ

Lösungsvorschlag:

Anzahl Blöcke wie in (a).

$2^{10} \times (32 \cdot 16 + 32 - 4 + 1)Bit = 553'984Bit$
Blöcke $\times$ (Datenbits + Adressbits − Word-Offset + Valid-Bit)

□

4.2: Cache Misses (maximal 13 Punkte)

Gegeben sei eine Matrix-Multiplikation einer 4x2 Matrix A mit einer 2x4 Matrix B. Das Resultat wird in einer 4x4 Matrix C gespeichert. Das folgende Pseudocode Programm berechnet diese Matrix-Multiplikation:

```
int A[4][2], B[2][4], C[4][4]; // Deklaration der Matrizen A, B und C

  .
  .  // Initialisierung von A und B
  .

for (i = 0; i < 4; i++){
    for (j = 0; j < 4; j++){
        C[j][i] = 0;
        for (k = 0; k < 2; k++){
            C[j][i] = A[j][k] * B[k][i] + C[j][i];
        }
    }
}
```

Es soll für dieses Programm die Anzahl der auftretenden Lese Cache-Misses bei der Matrix-Multiplikation berechnet werden. Der verwendete Cache hat acht Speicherblöcke, die jeweils ein Speicherwort (ein Matrixelement) fassen. Er ist 4-fach assoziativ und verwendet die Strategie "am längsten unbenutzt" (LRU), um Blöcke auszuwählen, die ersetzt werden sollen. Die Zählvariablen i, j und k werden in Registern gehalten. Der verwendete Cache sieht folgendermassen aus:

Set 0				Set 1			
Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7	Block 8

Gehen Sie davon aus, dass der Cache zu Beginn leer ist und ermitteln Sie die Zahl der Lese Cache-Misses **nur** für $i = 0,\ j \leq 1$. Gehen Sie ausserdem davon aus, dass bei der Berechnung der Addition/Multiplikation in der innersten Schleife die Lesezugriffe in folgender Reihenfolge erfolgen: A[j][k], B[k][i], C[j][i].

Die Matrizen sind wie folgt im Speicher abgebildet:

Adresse	Inhalt	Adresse	Inhalt	Adresse	Inhalt	Adresse	Inhalt
00000	A[0][0]	01000	B[0][0]	10000	C[0][0]	11000	C[2][0]
00001	A[0][1]	01001	B[0][1]	10001	C[0][1]	11001	C[2][1]
00010	A[1][0]	01010	B[0][2]	10010	C[0][2]	11010	C[2][2]
00011	A[1][1]	01011	B[0][3]	10011	C[0][3]	11011	C[2][3]
00100	A[2][0]	01100	B[1][0]	10100	C[1][0]	11100	C[3][0]
00101	A[2][1]	01101	B[1][1]	10101	C[1][1]	11101	C[3][1]
00110	A[3][0]	01110	B[1][2]	10110	C[1][2]	11110	C[3][2]
00111	A[3][1]	01111	B[1][3]	10111	C[1][3]	11111	C[3][3]

Verwenden Sie die folgende Tabelle für Ihre Lösung. In jeder Zeile ist dabei der aktuelle Cache-Status abgebildet. Die Zeitachse verläuft sequentiell von oben nach unten. Die ersten vier gelesenen Wörter sind zum besseren Verständnis bereits eingetragen.

i	j	k	gelesenes Wort	Set 0				Set 1			
0	0	0	A[0][0]								
			B[0][0]								
			C[0][0]								
0	0	1	A[0][1]								
0	1	0									
0	1	1									
Gesamtzahl Lese Misses:											

Lösungsvorschlag:

i	j	k	gelesenes Wort	Set 0				Set 1			
0	0	0	A[0][0]	A[0][0]							
			B[0][0]	A[0][0]	B[0][0]						
			C[0][0]	A[0][0]	B[0][0]	C[0][0]					
0	0	1	A[0][1]	A[0][0]	B[0][0]	C[0][0]		A[0][1]			
			B[1][0]	A[0][0]	B[0][0]	C[0][0]	B[1][0]	A[0][1]			
			C[0][0]	A[0][0]	B[0][0]	C[0][0]	B[1][0]	A[0][1]			
0	1	0	A[1][0]	A[1][0]	B[0][0]	C[0][0]	B[1][0]	A[0][1]			
			B[0][0]	A[1][0]	B[0][0]	C[0][0]	B[1][0]	A[0][1]			
			C[1][0]	A[1][0]	B[0][0]	C[0][0]	C[1][0]	A[0][1]			
0	1	1	A[1][1]	A[1][0]	B[0][0]	C[0][0]	C[1][0]	A[0][1]	A[1][1]		
			B[1][0]	A[1][0]	B[0][0]	B[1][0]	C[1][0]	A[0][1]	A[1][1]		
			C[1][0]	A[1][0]	B[0][0]	B[1][0]	C[1][0]	A[0][1]	A[1][1]		
Gesamtzahl Lese Misses: 9											

□

4.3: Performance (maximal 12 Punkte)

Gegeben ist ein Rechner mit den folgenden Spezifikationen:

- Instruktions-Missrate 2%
- Daten-Missrate 4%
- CPI eines perfekten Caches (Missrate 0%): 2
- Miss-Strafe: 50 Zyklen pro Miss
- Ein Drittel aller Instruktionen sind Speicherinstruktionen

Um welchen Faktor muss die Taktfrequenz erhöht werden, damit sich die Geschwindigkeit des Gesamtsystems (a) um 50% erhöht und (b) verdoppelt? Dies unter der Annahme, dass sich die Missraten und die Speicherzugriffszeit nicht ändern.

Lösungsvorschlag: Zuerst wird die CPI mit stalls für die originale Taktfrequenz berechnet:
Instruktions-Misszyklen pro Instruktion = $0.02 \cdot 50$
Daten-Misszyklen pro Instruktion = $0.04 \cdot 50 \cdot 1/3$
Memory Stall cycles = $0.02 \cdot 50 + 0.04 \cdot 50 \cdot 1/3$
CPI_s mit stalls = $CPI + 0.02 \cdot 50 + 0.04 \cdot 50 \cdot 1/3 = 3.67$

Nun wird die CPI mit stalls für die verschnellerte Taktfrequenz (in Abhängigkeit vom Faktor x, mit welchem die Taktfrequenz beschleunigt wird) berechnet:
Neue Miss-Strafe: $x \cdot 50$ Zyklen pro Miss
CPI_f mit stalls = $CPI + 0.02 \cdot 50x + 0.04 \cdot 50x \cdot 1/3$

Nun kann die Beschleunigung des Systems s berechnet werden ($I =$ Anzahl Instruktionen, $T =$ Ausführzeit eines Zyklus):

$$
\begin{aligned}
s &= \frac{\text{Ausführungszeit mit langsamer Taktfrequenz}}{\text{Ausführungszeit mit schneller Taktfrequenz}} \\
&= \frac{I \cdot CPI_s \cdot T}{I \cdot CPI_f \cdot T/x} \\
&= \frac{CPI_s}{CPI_f/x} \\
&= \frac{CPI_s}{\frac{CPI + 0.02 \cdot 50x + 0.04 \cdot 50x \cdot 1/3}{x}} \\
&= \frac{CPI_s}{CPI/x + 0.02 \cdot 50 + 0.04 \cdot 50 \cdot 1/3} \\
\frac{CPI_s}{s} &= \frac{CPI}{x} + 0.02 \cdot 50 + 0.04 \cdot 50 \cdot 1/3 \\
\frac{CPI_s}{s} - 0.02 \cdot 50 - 0.04 \cdot 50 \cdot 1/3 &= \frac{CPI}{x} \\
x &= \frac{CPI}{CPI_s/s - 0.02 \cdot 50 - 0.04 \cdot 50 \cdot 1/3}
\end{aligned}
$$

(a) $s = 1.5 \rightarrow x = 2.5714$

(b) $s = 2 \rightarrow x = 12$

□

Aufgabe 5 : Assembler und Pipelining (maximal 29 Punkte)

5.1: Assemblerprogrammierung (maximal 22 Punkte)

Das Spiel *Türme von Hanoi* besteht aus drei Stapeln und n Scheiben paarweise verschiedener Grösse. Zu Beginn liegen alle Scheiben auf Stapel 1 (vgl. Abbildung 1), mit der grössten Scheibe unten und der kleinsten oben. Ziel des Spiels ist es, alle Scheiben von 1 nach 3 zu versetzen. Bei jedem Zug darf die oberste Scheibe eines beliebigen Stapels auf einen der anderen gelegt werden, vorausgesetzt dort liegt nicht schon eine kleinere Scheibe.

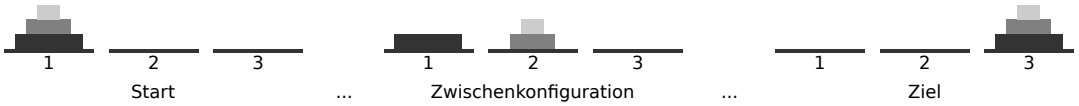


Abbildung 1: Start, Ziel und eine Zwischenkonfiguration für $n = 3$ Scheiben.

Das folgende MIPS Assemblerprogramm berechnet für eine gegebene Scheibenanzahl den Lösungsweg mit der minimalen Anzahl an Spielzügen, wobei die Prozedur `print` jeden Zug in der Form `move $a0 -> $a1` ausgibt (z. B. bedeutet `move 1 -> 3`, dass die oberste Scheibe von Stapel 1 nach Stapel 3 zu legen ist):

```
(00) hanoi:  addi  $sp,$sp,-20
(01)         sw    $ra,16($sp)
(02)         sw    $a3,12($sp)
(03)         sw    $a2,8($sp)
(04)         sw    $a1,4($sp)
(05)         sw    $a0,0($sp)
(06)         slt   $t0,$zero,$a0
(07)         bne   $t0,$zero,L1
(08)         addi  $sp,$sp,20
(09)         jr    $ra
(10) L1:     addi  $a0,$a0,-1
(11)         lw    $a2,12($sp)
(12)         lw    $a3,8($sp)
(13)         jal   hanoi
(14)         lw    $a0,4($sp)
(15)         lw    $a1,8($sp)
(16)         jal   print
(17)         lw    $a0,0($sp)
(18)         addi  $a0,$a0,-1
(19)         lw    $a1,12($sp)
(20)         lw    $a2,8($sp)
(21)         lw    $a3,4($sp)
(22)         jal   hanoi
(23)         lw    $ra,16($sp)
(24)         addi  $sp,$sp,20
(25)         jr    $ra
```

(a) Übersetzen Sie das Assemblerprogramm in eine C-Funktion. Die Ausgabe der Spielzüge soll durch eine geeignete `printf` Anweisung realisiert werden. Zur Vereinfachung sind die Argumente der C-Funktion bereits gegeben. Die Parameterübergabe im Assembler-Programm erfolgt mittels der vier Argumentregister `$a0`–`$a3`.

```
void hanoi(int n, int from, int to, int using) {
    ...
}
```

Lösungsvorschlag: Die gesuchte C-Funktion sieht folgendermassen aus:

```
void hanoi(int n, int from, int to, int using) {
    if(n > 0) {
        hanoi(n-1, from, using, to);
        printf("move %d -> %d\n", from, to);
        hanoi(n-1, using, to, from);
    }
}
```

□

(b) Für $n = 3$ Scheiben ermittelt das Assemblerprogramm den folgenden Lösungsweg:

```
move 1 -> 3
move 1 -> 2
move 3 -> 2
move 1 -> 3
move 2 -> 1
move 2 -> 3
move 1 -> 3
```

Geben Sie für diesen Fall den minimalen Wert des Stack Pointers `$sp` an, vorausgesetzt `$sp` hat vor Ausführung der ersten Assembleranweisung den Wert 1000 (dezimal).

Lösungsvorschlag: Der minimale Wert von `$sp` ist durch die maximale Rekursionstiefe bestimmt, welche wiederum von der Scheibenanzahl n abhängt. Nach dem ersten Aufruf der Prozedur `hanoi` wird diese für $n = 3$ weitere drei Mal rekursiv aufgerufen bevor der Stack (zunächst) wieder abgebaut wird. Der minimale Wert des Stack Pointers `$sp` ist demnach 920.

□

5.2: Pipelining (maximal 7 Punkte)

Gegeben sei ein Prozessor mit der aus der Vorlesung bekannten fünfstufigen Pipeline Architektur (IF-ID-EX-MEM-WB). Forwarding wird *nicht* unterstützt.

(a) Geben Sie unter Verwendung von Zeilennummern und Registernamen die Datenabhängigkeiten an, die bei der Ausführung des nachfolgenden Assemblerprogramms auf diesem Prozessor zu Data Hazards führen (z.B. bedeutet `(13)-(11) $s1`, dass die Anweisung in Zeile `(13)` von der in `(11)` bzgl. des Registers `$s1` abhängt). Fügen Sie anschliessend eine minimale Anzahl von `nop`s ein, um das Problem zu beheben.

```
(01)        ori   $t0,$zero,3
(02)        ori   $t1,$zero,4
(03)        sub   $t1,$t0,$t1
(04)        ori   $t0,$zero,10
(05)        add   $t1,$t0,$t1
```

Lösungsvorschlag: Es gibt vier Datenabhängigkeiten, die zu Data Hazards führen:

```
(03)-(01) $t0
(03)-(02) $t1
(05)-(03) $t1
(05)-(04) $t0
```

Durch Einfügen von minimal vier `nop`s können diese behoben werden:

```
(01)        ori   $t0,$zero,3
(02)        ori   $t1,$zero,4
            nop
            nop
(03)        sub   $t1,$t0,$t1
(04)        ori   $t0,$zero,10
            nop
            nop
(05)        add   $t1,$t0,$t1
```

□

(b) Eliminieren Sie den Data Hazard im Assemblerprogramm aus Teilaufgabe 5.1 durch Umsortieren der Zeilen `(17)`-`(22)`, wobei das Ergebnis des Programms nicht verändert werden soll:

```
(17)        lw    $a0,0($sp)
(18)        addi  $a0,$a0,-1
(19)        lw    $a1,12($sp)
(20)        lw    $a2,8($sp)
(21)        lw    $a3,4($sp)
(22)        jal   hanoi
```

Lösungsvorschlag: Um den Data Hazard bzgl. `$a0` zu eliminieren, müssen zwischen den Anweisungen `lw $a0,0($sp)` und `addi $a0,$a0,-1` mindestens zwei weitere `lw`s ausgeführt werden. Bspw. könnte man folgendermassen umsortieren:

```
(17)        lw    $a0,0($sp)
(18)        lw    $a1,12($sp)
(19)        lw    $a2,8($sp)
(20)        addi  $a0,$a0,-1
(21)        lw    $a3,4($sp)
(22)        jal   hanoi
```

□

Aufgabe 6 : Datenpfad (maximal 26 Punkte)

6.1: Verständnisfragen (maximal 6 Punkte)

(a) In Abbildung 2 sehen Sie den Datenpfad eines Prozessors. Handelt es sich bei dem dargestellten Datenpfad um eine Einzyklen-Implementierung oder eine Pipeline-Architektur? Beschreiben Sie in maximal drei Sätzen den grundlegenden architekturellen Unterschied zwischen den beiden genannten Varianten. [2 Punkte]

Lösungsvorschlag: Bei dem dargestellten Datenpfad handelt es sich um eine Pipeline-Architektur.

Durch die Trennung des Pfades in mehrere Pipeline-Stufen werden an den Schnittstellen neue Zwischenspeicher benötigt. Diese sind in Abbildung 2 durch die vertikalen Balken angedeutet. (Hennessy&Patterson: Instruction register, Memory data register, A, B, ALUOut). Zusätzlich müssen Multiplexer ergänzt bzw. erweitert werden.

□

(b) Der abgebildete Datenpfad enthält zwei Addierer (markiert durch A und B). Beschreiben Sie die Funktion der beiden Addierer in Bezug auf den Ablauf des Datenpfades. Welche Instruktionen können nicht mehr korrekt ausgeführt werden, wenn Addierer B fehlt? [2 Punkte]

Lösungsvorschlag: Addierer A wird zur Erhöhung des Program Counters (PC) um ein Wort (32bit) benötigt. Dadurch wird die nächste Instruktion des 32bit breiten Befehlssatzes aus dem Instruktionsspeicher geladen.

Addierer B ist zur Implementierung von Verzweigungsbefehlen (z.B. $\boldsymbol{beq}$) (mit indirekt gegebenen Sprungzielen) notwendig. Diese können ohne Addierer B nicht mehr korrekt ausgeführt werden.

□

(c) Der Prozessor befindet sich bei der Programmausführung an der Adresse $\mathbf{104}$ (dezimal) des Instruktionsspeichers. Im Register $\mathbf{\$3}$ befindet sich der Wert $\mathbf{0}$, die Instruktion $\boldsymbol{beq}\ \mathbf{\$3, \$0, 8}$ (Worte) wird ausgeführt. Welche Adresse des Instruktionsspeichers wird der Prozessor nach der EX-Stufe der beschriebenen Instruktion laden? [2 Punkte]

Lösungsvorschlag: Das Register $\mathbf{\$0}$ ist fest mit dem Wert $\mathbf{0}$ verdrahtet. Der Verzweigungsbefehl hat daher das Ergebnis “taken”.

Nach der Verzweigung wird daher die Adresse $\boldsymbol{PC}_{\boldsymbol{new}} = \mathbf{104 + 4 + 4 \times 8 = 140}$ (dezimal) geladen.

□

6.2: Implementierung einer neuen Instruktion (maximal 20 Punkte)

In dieser Aufgabe soll der gegebene Datenpfad (Abbildung 2) um die Instruktion **m4a** (multiply-by-four-and-accumulate) erweitert werden. Diese neue Instruktion führt die Berechnung $\boldsymbol{A = B + C \times 4}$ durch.

Neue Instruktion

$\boldsymbol{m4a \quad rs, rt, immed}$

Register Transfers

$\boldsymbol{Reg[rt] \leftarrow Reg[rs] + sign\_extend(immed) \times 4;}$

Kodierung

31	25	20	15	10	5	0
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	
$m4a$	rs	rt	$immed$			

(a) Modifizieren Sie den abgebildeten Datenpfad und die Steuerung des Prozessors so, dass die Instruktion $m4a$ implementiert wird. Tragen Sie alle nötigen Änderungen bzgl. des Datenpfades in Abbildung 2 ein. Sie benötigen dazu keine weiteren Einheiten. Modifizieren Sie falls nötig lediglich vorhandene Multiplexer-Einheiten und fügen Sie eventuell benötigte Steuer- und Datenleitungen hinzu. Die Fähigkeiten der ALU-Einheit sind auf den aus der Vorlesung bekannten Umfang beschränkt.
Geben Sie ausserdem die Belegung aller Steuerungssignale in der nachfolgenden Tabelle an. Hierbei sollen Steuerungssignale, deren Wert für die Ausführung der Instruktion $m4a$ nicht relevant ist, mit "-" gekennzeichnet werden. [20 Punkte]

Instr.	EX Stage Control Lines				MEM Stage Control Lines			WB Stage Control Lines	
	RegDst	ALUOp1	ALUOpo	ALUSrc	PCSrc	MemRead	MemWrite	RegWrite	MemtoReg
$m4a$									

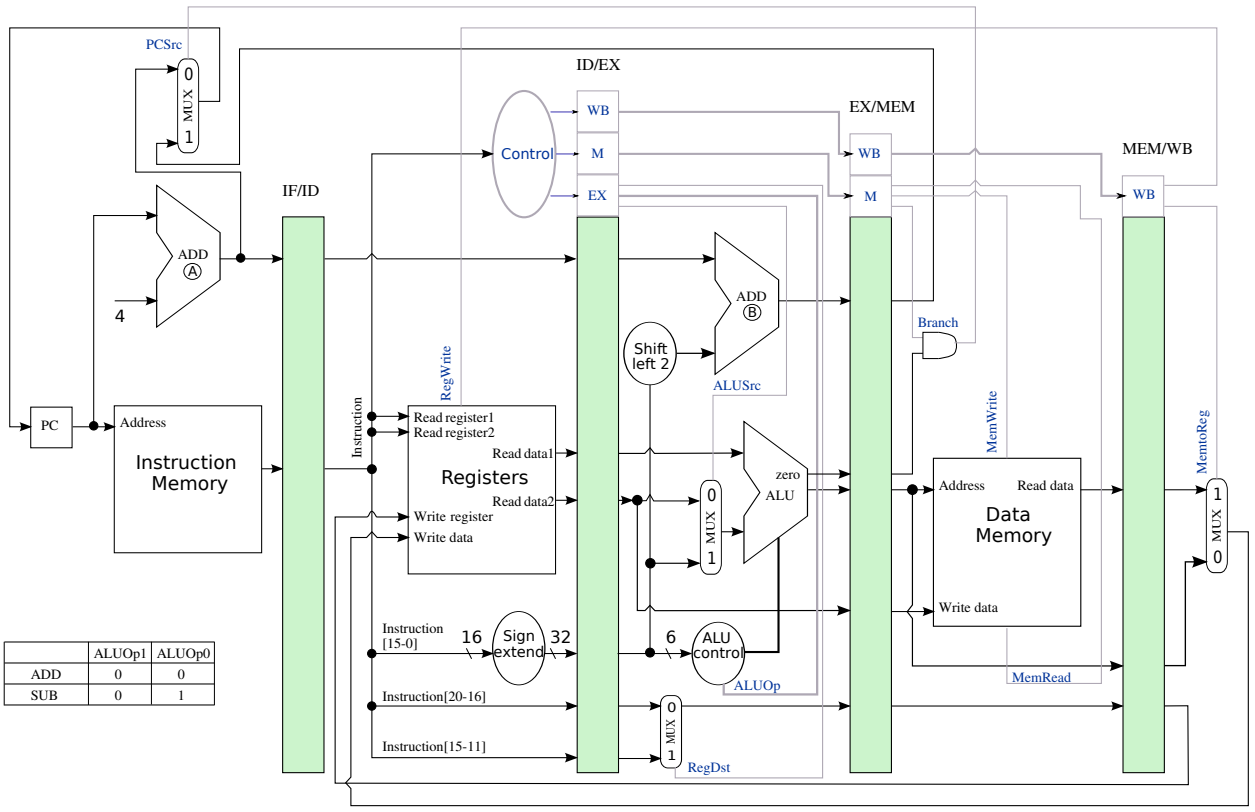
Lösungsvorschlag:

Instr.	EX Stage Control Lines				MEM Stage Control Lines			WB Stage Control Lines	
	RegDst	ALUOp1	ALUOpo	ALUSrc	PCSrc	MemRead	MemWrite	RegWrite	MemtoReg
$m4a$	0	0	0	**2**	0	-	0	1	0

□

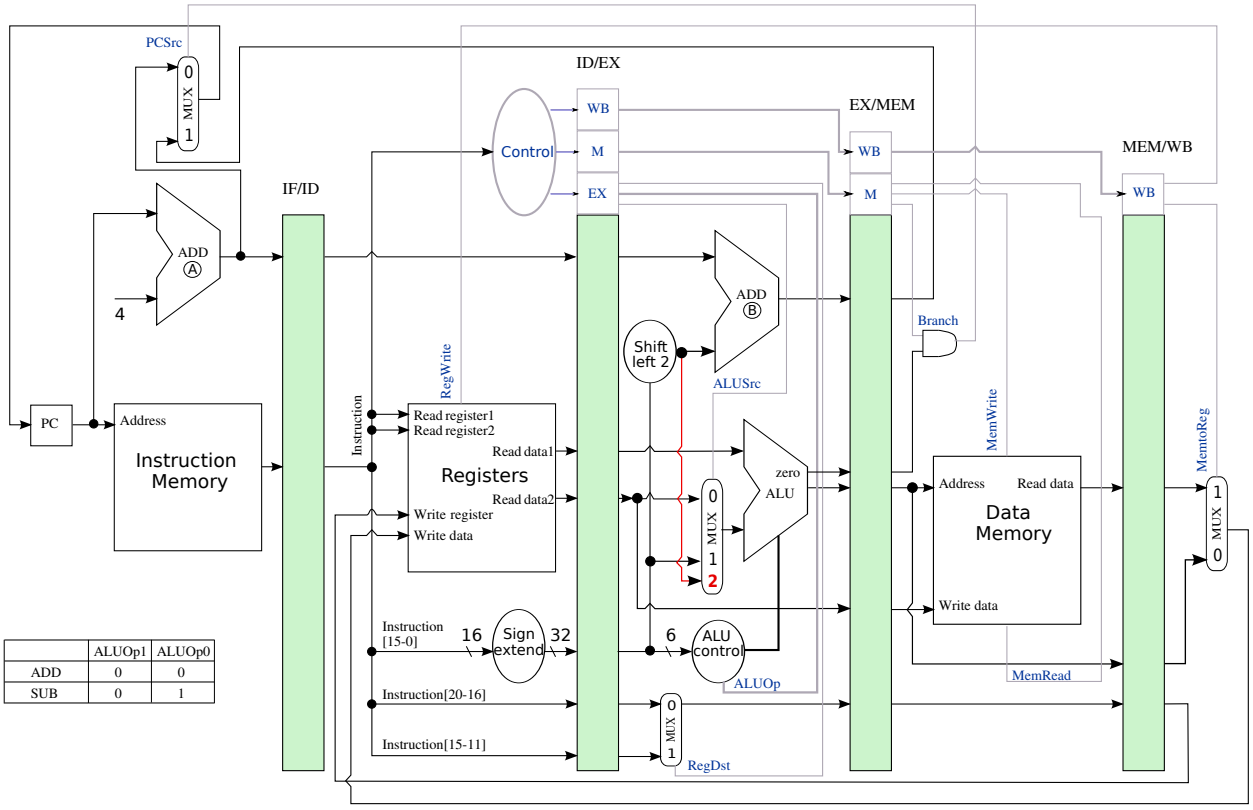
Lösungsvorschlag:

□



	ALUOp1	ALUOp0
ADD	0	0
SUB	0	1

Abbildung 1: Datenpfad, Kontrolleinheiten und Steuersignale der CPU.



	ALUOp1	ALUOp0
ADD	0	0
SUB	0	1

Abbildung 2: Datenpfad, Kontrolleinheiten und Steuersignale der CPU.