Beispielhafte Prüfungsaufgaben zur Vorlesung TI I, gestellt im Frühjahr 2009

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

Aufgabe 4 : “Pipelined” Daten- und Kontrollpfad (maximal 25 Punkte)

Gegeben sei die neue MIPS Instruktion *lwu* (load word - update). Diese Instruktion hat die selbe Funktion wie die übliche *lw*-Instruktion, führt aber zusätzlich eine Aktualisierung des Registers *rs* durch. Im Folgenden sind die Registerfeldoperationen sowie die Kodierung von *lwu* dargestellt:

Neue Instruktion

lwu rt, immed(rs)

Register Transfers

address ← Reg[rs] + signExt(immed);
Reg[rt] ← mem[address];
Reg[rs] ← address;

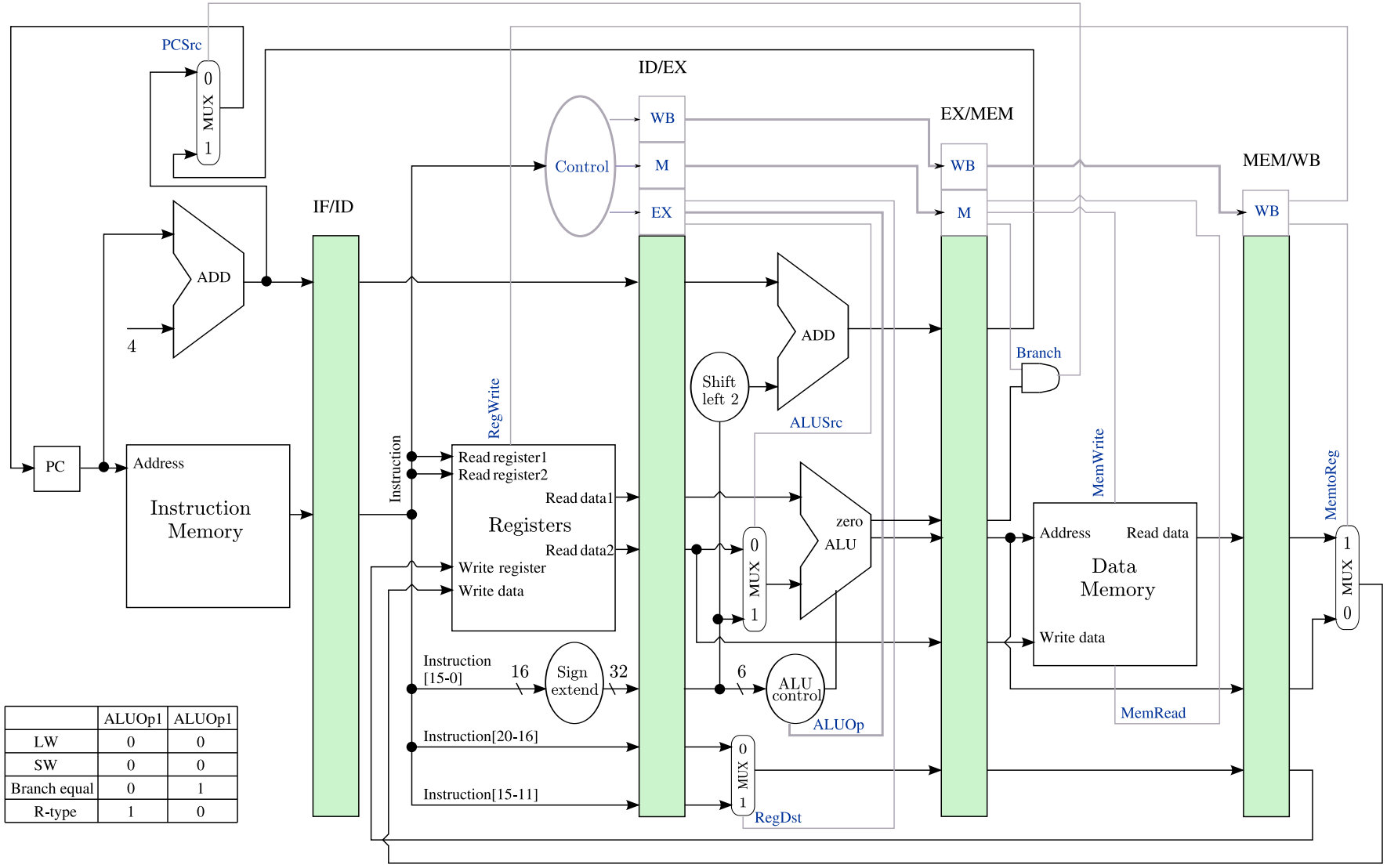
Kodierung

31 – 26	25 – 21	20 – 16	15 – 11	10 – 6	5 – 0
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
lwu	**rs**	**rt**	**immed**		

4.1: Entwurf eines “Pipelined” Prozessors (maximal 15 Punkte)

Modifizieren sie den Datenpfad und die Steuerung des Pipeline-Prozessors so, dass die Instruktion *lwu* implementiert wird. Beachten Sie dabei, dass bei einer Pipeline-Implementierung das Schreiben von Registern innerhalb der WB-Phase stattfinden soll. Um dies zu gewährleisten, soll das Registerfeld um einen zweiten Schreibzugang (“write-port”) erweitert werden, so dass ein Schreibzugriff auf 2 Register gleichzeitig stattfinden kann. Tragen Sie alle nötigen Änderungen bzgl. des Datenpfades in die Skizze auf der nächsten Seite ein. Geben Sie ausserdem alle Steuerungssignale in der nachfolgenden Tabelle an.

Instr.	**EX Stage Control Lines**					**MEM Stage Control Lines**				**WB Stage control Lines**		
	RegDst	ALUOp1	ALUOpo	ALUSrc		PCSrc	MemRead	MemWrite		RegWrite	MemtoReg	
lwu												



	ALUOp1	ALUOp1
LW	0	0
SW	0	0
Branch equal	0	1
R-type	1	0

Abbildung 1: Datenpfad, Kontrolleinheiten und Steuersignale der Pipline-CPU.

Lösungsvorschlag: Siehe angepassten Daten- und Kontrollpfad in Abbildung 2.

Instr.	EX Stage Control Lines					MEM Stage Control Lines				WB Stage control Lines		
	RegDst	ALUOp1	ALUOpo	ALUSrc		PCSrc	MemRead	MemWrite		RegWrite	MemtoReg	Reg. Wr. 2
lwu	0	0	0	1		0	1	0		1	1	1

□

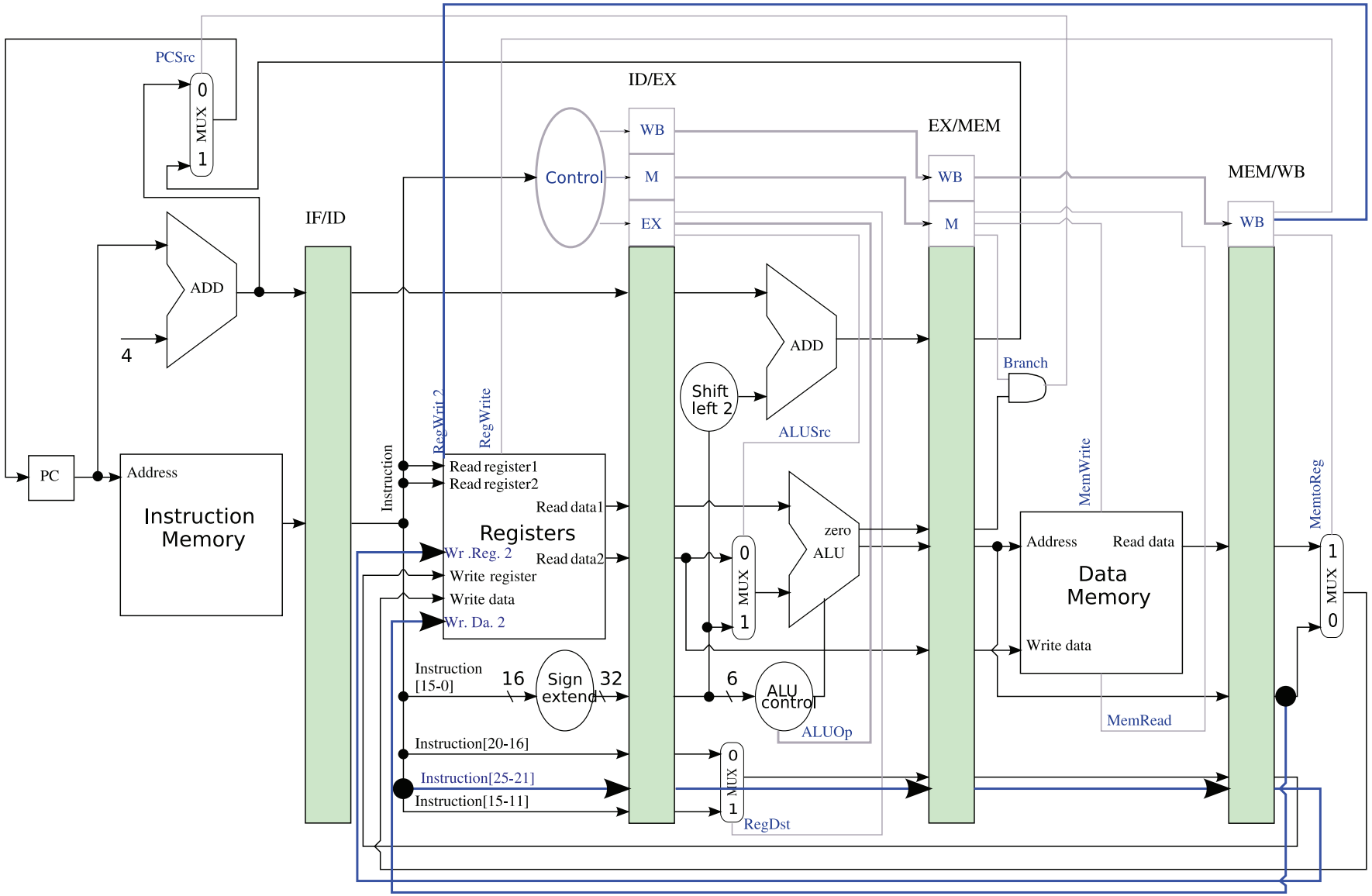


Abbildung 2: Lösungsvorschlag

4.2: "Timing" eines "Pipelined" Prozessors (maximal 10 Punkte)

Nehmen Sie an, dass die Verzögerungen der einzelnen Komponenten des Pipeline- Datenpfades aus Abb. 1 wie folgt charakterisiert sind:

Verzögerung von ALU und Hauptspeicher: $2\,ns$;
Lese- oder Schreibzugriff auf das Registerfeld: $1\,ns$.

(a) Die anderen Komponenten sind als verzögerungsfrei anzunehmen. Wie gross ist die minimale Taktperiode (clock period), so dass der Prozessor-Entwurf fehlerfrei funktioniert?

[1 Punkt]

Lösungsvorschlag: Längster Verzögerungspfad zwischen ID/EX und EX/MEM durch ALU: 2ns

□

(b) Nehmen Sie nun an, dass zusätzlich zu den obigen Verzögerungen, die Verzögerung der Multiplexer bei $0.1\,ns$ liegt. Was ist nun die minimale Taktperiode, so dass der Entwurf fehlerfrei funktioniert? Welche Stufe beschränkt hierbei die Ausführungszeit nach unten? [2 Punkte]

Lösungsvorschlag: Längster Verzögerungspfad zwischen ID/EX und EX/MEM Phase über Multiplexer und ALU: 2.1ns

□

(c) Die Modifikation des Registerfiles wie in Aufgabe **4.1** beschrieben erhöhe die Verzögerung der entsprechenden Registerfeldoperationen von 1 ns auf 2.1 ns. Nehmen Sie an, dass wir ein Programm mit 1000 Instruktionen ausführen wollen, das keine "Hazards" oder "Stalls" verursacht. Wie viele **lwu** Instruktionen gibt es in diesem Programm mindestens, wenn die Programmausführung schneller als im urspünglichen Entwurf (siehe Abb. 1) ist? Gehen Sie davon aus, dass der Multiplexer keine Verzögerung verursacht. [7 Punkte]

Hinweis: Der ursprüngliche Prozessor-Entwurf benötigt 2 Instruktionen, um die gleiche Funktionalität einer *lwu* Instruktion zu erzielen.

Lösungsvorschlag:

$P_{alt} = 2ns$ (alte Taktperiode)

$P_{neu} = 2.1ns$ (neue Taktperiode)

$N = 1000$ (Zahl der Instruktionen)

x (Anzahl der **lwu**-Instruktionen)

$N \cdot P_{neu} < (N + x) \cdot P_{alt}$

$1000 \cdot 2.1 < (1000 + x) \cdot 2 => x > 50$

□

Aufgabe 5 : Assembler und Pipelining (maximal 38 Punkte)

5.1: Assemblerprogrammierung (maximal 17 Punkte)

Gegeben sei folgende Funktion in C, die n^i berechnet:

```
(1)int power(int n, int i) {
(2)  if( i == 0) {
(3)    return 1;
(4)  }  else {
(5) i = i-1;
(5)     return n * power(n, i);
(6)  }
(7)}
```

(Die Zahlen in Klammern geben die Zeilennummern an.)

(a) (15 Punkte) Übersetzen Sie das Programm in Assembler unter Beibehaltung der rekursiven Struktur. Zur Durchführung von Multiplikationen wurde der MIPS Instruktionssatz um die Instruktion `mul` erweitert (z.B.: `mul $s1,$s2,$s3` entspricht $\$s1 = \$s2 \cdot \$s3$). Übergeben Sie die Funktionsargumente mittels der hierfür vorgesehenen Register.

Lösungsvorschlag:

```
power:  addi $sp, $sp, -4     (Platz für ein Reg. auf Stack)
        sw   $ra, 0($sp)      (Ruecksprungadresse sichern)
        bne  $a1, $zero, L1   (if i <> 0 jump to L1)
        addi $v0, $zero, 1    (return 1)
        j    exit
L1:     addi $sp, $sp, -4     (Platz für ein Reg. auf Stack)
        sw   $a0, 0($sp)      (Parameter n sichern)
        sub  $a1, $a1, 1      (decrement)
        jal  power
        lw   $a0, 0($sp)      (Parameter n wiederherstellen)
        addi $sp, $sp, 4
        mul  $v0, $a0, $v0    (n * power(n, i-1))
exit:   lw   $ra, 0($sp)      (Ruecksprungadresse wiederherstellen)
        addi $sp, $sp, 4
        jr   $ra
```

□

(b) (2 Punkte) Beschreiben Sie für Ihr Assemblerprogramm die Grösse des Stacks als Funktion des Argumentes i.

Lösungsvorschlag: $2i + 1$

□

5.2: Pipelines und Optimierung (maximal 21 Punkte)

Gegeben sei eine 5-stufige Prozessor-Pipeline, wie in der Vorlesung vorgestellt, mit den Stufen:

- Laden der Instruktion (IF)
- Dekodierung der Instruktion und Lesen der Register (ID)
- Ausführung (EX)
- Datenspeicher-Zugriff (MEM)
- Speichern der Resultate (WB).

Hazards werden grundsätzlich durch Stalls vermieden. Sie werden eingefügt, sobald am Ende der IF-Stufe ein Hazard detektiert wird. Forwarding wird nicht unterstützt. Register, die von der WB-Stufe beschrieben werden, können im gleichen Zyklus von der ID-Stufe gelesen werden. Bei Verzweigungen wird die statische Sprungvorhersage "not taken" verwendet. Verzweigungsadressen werden in der ID-Stufe berechnet.

(a) (10 Punkte) Gegeben sei folgendes Assemblerprogramm:

```
(1)        lw    $t1, 0($a0)
(2)        lw    $t2, 4($a0)
(3)        lw    $t3, 8($a0)
(4)        add   $v0, $zero, $zero
(5)        add   $t5, $t1, $t2
(6) test:  slti  $t4, $t3, 1
(7)        beq   $t4, $zero, loop
(8)        jr    $ra
(9) loop:  add   $t5, $t3, $t5
(10)       add   $v0, $t5, $zero
(11)       subi  $t3, $t3, 1
(12)       j     test
```

Tragen Sie die zeitliche Belegung der gegebenen Pipeline in Tabelle 1 ein. Nehmen Sie dazu an, dass das Argument, welches in Register `$t3` geladen wird, den Wert 1 hat. Um sich auf die einzelnen Zeilen im Assemblercode zu beziehen, verwenden Sie die in Klammern angegebenen Zeilennummern.

Lösungsvorschlag: : siehe 2. Tabelle

□

(b) (8 Punkte) Optimieren Sie das Programm so, dass möglichst wenige Prozessorzyklen notwendig sind. Dabei dürfen Instruktionen umsortiert, eingefügt, gelöscht oder verändert werden. Nehmen Sie für die Optimierung an, dass das Argument, welches in Register `$t3` geladen wird, jeden beliebigen Wert annehmen kann. Behalten Sie die Funktionalität des Programmes bei. Funktionen sind äquivalent, wenn für gleiche Eingangsargumente der Rückgabewert übereinstimmt.

Lösungsvorschlag: : siehe auch 3. Tabelle

```
(1)         lw    $t3, 8($a0)
(2)         lw    $t2, 4($a0)
(3)         lw    $t1, 0($a0)
(4)         slti  $t4, $t3, 1
(5)         bne   $t4, $zero, npos
(6)         add   $v0, $t1, $t2
(7)  loop:  add   $v0, $v0, $t3
(8)         subi  $t3, $t3, 1
(9)         bne   $t3, $zero, loop
(10)        j     exit
(11) npos:  add   $v0, $zero, $zero
(12) exit:  jr    $ra
```

□

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
(1)	IF	ID	EX	MEM	WB																											

Tabelle 1: Pipelining Diagramm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
(1)	IF	ID	E	M	W																														
(2)		IF	ID	E	M	W																														
(3)			IF	ID	E	M	W																													
(4)				IF	ID	E	M	W																												
(5)					IF	ID	E	M	W																											
(6)						IF	ID	E	M	W																										
(7)							IF	-	-	ID	E	M	W																							
(8)										IF																									
(9)											IF	ID	E	M	W																				
(10)												IF	-	-	ID	E	M	W																	
(11)															IF	ID	E	M	W																
(12)																IF	ID	E	M	W															
(13)																	IF																		
(6)																		IF	ID	E	M	W													
(7)																			IF	-	-	ID	E	M	W										
(8)																						IF	ID	E	M	W									

Tabelle 2: Pipelining Diagramm - Lösung zu Aufgabenteil 5.2 (a)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
(1)	IF	ID	E	M	W																										
(2)		IF	ID	E	M	W																										
(3)			IF	ID	E	M	W																								
(4)				IF	ID	E	M	W																							
(5)					IF	-	-	ID	E	M	W																				
(6)								IF	ID	E	M	W																			
(7)									IF	-	-	ID	E	M	W																
(8)												IF	ID	E	M	W															
(9)													IF	-	-	ID	E	M	W												
(10)																IF	ID	E	M	W											
(11)																	IF														
(12)																		IF	ID	E	M	W									

Tabelle 3: Pipelining Diagramm - optimierte Lösung zu Aufgabenteil 5.2 (b)

(c) (3 Punkte) Gegeben sei das folgende Assemblerprogramm:

```
(1)       add    $s0, $t0, $t1
(2)       sub    $t2, $s0, $t3
```

Stellen Sie die zeitliche Belegung der gegebenen Pipeline dar, unter der Annahme, dass Forwarding unterstützt wird. Verwenden Sie dazu Tabelle 4.
Ersetzen Sie nun die Instruktion in Zeile (1) durch `lw $s0, 4($t1)`. Stellen Sie die neue zeitlich Belegung der Pipeline in Tabelle 5 dar. Geben Sie jeweils die Anzahl der Stalls an.

	1	2	3	4	5	6	7	8

Tabelle 4: Pipelining Diagramm

	1	2	3	4	5	6	7	8

Tabelle 5: Pipelining Diagramm

Lösungsvorschlag:

	1	2	3	4	5	6	7	8
(1)	IF	ID	E	M	W			
(2)		IF	ID	E	M	W		

Tabelle 6: Pipelining Diagramm
Keine Stalls, Forwarding zwischen `EX` in (1) und `EX` in (2)

	1	2	3	4	5	6	7	8
(1)	IF	ID	E	M	W			
(2)		IF	-	ID	E	M	W	

Tabelle 7: Pipelining Diagramm
Ein Stall, Forwarding zwischen `MEM` in (1) und `EX` in (2)

□

Aufgabe 6 : Cache (maximal 27 Punkte)

6.1: Cache Hierarchie (maximal 10 Punkte)

Gegeben sei ein Rechner mit folgender Basis-Spezifikation:

- Prozessor: Taktrate: $f = 2\,\text{GHz}$, CPI bei perfektem L1-Cache: $CPI_{perfekt}$ = 2.5
- L1-Cache: Miss-Rate: $L1_{miss} = 5\%$
- Hauptspeicher: Zugriffszeit: $T_{mem} = 10\,\text{ns}$

Es gibt nun die folgenden Möglichkeiten, um den Rechner zu beschleunigen:

(1) Erhöhung der Taktrate um 25%, wobei die Zugriffszeit auf den Hauptspeicher gleich bleibt, d.h. die Anzahl der Miss-Zyklen vergrössert sich entsprechend.

(2) Vergrösserung des L1-Caches, wobei die Miss-Rate von 5% auf 3% sinkt: $L1'_{miss} = 3\%$.

(3) Verwendung eines L2-Caches bei gleichbleibendem L1-Cache, d.h. $CPI_{perfekt}$ = 2.5, $L1_{miss} = 5\%$. Die Miss-Rate des L2-Caches ist $L2_{miss} = 10\%$. Die Speicherzugriffszeiten bei einem L1-Cache Miss sind wie folgt gegeben:

- L1 Miss, L2 Hit: $T_{L2} = 2\,\text{ns}$
- L1 Miss, L2 Miss: $T_{mem,\,L2} = 12\,\text{ns}$

Beantworten Sie die folgenden Fragen:

(a) (3 Punkte) Wieviel Zeit benötigt im Durchschnitt die Ausführung einer Instruktion in der Basis-Spezifikation?

(b) (4 Punkte) Wieviel Zeit benötigt im Durchschnitt die Ausführung einer Instruktion, wenn je eine der drei Möglichkeiten ausgenutzt wird?

(c) (3 Punkte) Wieviel Zeit benötigt im Durchschnitt die Ausführung einer Instruktion, wenn alle drei Möglichkeiten gleichzeitig ausgenutzt werden?

Lösungsvorschlag:
$f = 2\,\text{GHz} \rightarrow T = 0.5\,\text{ns}$ pro Zyklus bzw. $f_{+25\%} = 2.5\,\text{GHz} \rightarrow T_{neu} = 0.4\,\text{ns}$ pro Zyklus

(a) $T_0 = CPI_{perfekt} \cdot T + L1_{miss} \cdot T_{mem} = 2.5 \cdot 0.5\,ns + 0.05 \cdot 10\,ns = 1.75\,ns$

(b) $T_1 = CPI_{perfekt} \cdot T_{neu} + L1_{miss} \cdot T_{mem} = 2.5 \cdot 0.4\,ns + 0.05 \cdot 10\,ns = 1.5\,ns$
$T_2 = CPI_{perfekt} \cdot T + L1_{miss} \cdot T_{mem} = 2.5 \cdot 0.5\,ns + 0.03 \cdot 10\,ns = 1.55\,ns$
$T_3 = CPI_{perfekt} \cdot T + L1_{miss} \cdot ((1 - L2_{miss}) \cdot T_{L2} + L2_{miss} \cdot T_{mem,\,L2}) = 2.5 \cdot 0.5\,ns + 0.05 \cdot (0.9 \cdot 2\,ns + 0.1 \cdot 12\,ns) = 1.4\,ns$

(c) $T_{123} = CPI_{perfekt} \cdot T_{neu} + L1_{miss} \cdot ((1 - L2_{miss}) \cdot T_{L2} + L2_{miss} \cdot T_{mem,\,L2}) = 2.5 \cdot 0.4\,ns + 0.03 \cdot (0.9 \cdot 2\,ns + 0.1 \cdot 12\,ns) = 1.09\,ns$

6.2: Cache Organisation (maximal 17 Punkte)

Wir betrachten einen 8-bit Prozessor mit einem Speicher von 256 Bytes, der byte-weise adressiert wird. Für diesen Prozessor werden vier aus der Vorlesung bekannte Caches betrachtet:

(1) direkte Abbildung (direct-mapped)

(2) teilassoziativ mit 2 Einträgen pro Index (2-way set associative)

(3) teilassoziativ mit 4 Einträgen pro Index (4-way set associative)

(4) vollassoziativ (fully associative)

Die Caches sind wie folgt spezifiziert:

- Wortgrösse: 1 Byte
- Blockgrösse: 2 Bytes
- Cachegrösse: 8 Blöcke
- Blockersetzungsschema: LRU-Verfahren (least recently used), d.h. der am längsten unbenutzte Block wird ersetzt

(a) (3 Punkte) Geben Sie für alle vier Caches die Aufteilung einer Speicheradresse in Offset, Index und Tag an.

Lösungsvorschlag:

- direct-mapped: offset: 1 bit, index: 3 bits, tag: 4 bits
- 2-way set associative: offset: 1 bit, index: 2 bits, tag: 5 bits
- 4-way set associative: offset: 1 bit, index: 1 bit, tag: 6 bits
- fully associative: offset: 1 bit, index: 0 bits, tag: 7 bits

(b) (4 Punkte) Geben Sie für alle vier Caches die effektive Grösse in Bits an. Berücksichtigen Sie dazu lediglich die Bits für Daten und Tags.

Lösungsvorschlag:
#Bits = #Blöcke · (#Bits(Block) + #Bits(Tag))

- direct-mapped: $8 \cdot (2 \cdot 8 + 4) = 160$
- 2-way set associative: offset: $8 \cdot (2 \cdot 8 + 5) = 168$
- 4-way set associative: offset: $8 \cdot (2 \cdot 8 + 6) = 176$
- fully associative: offset: $8 \cdot (2 \cdot 8 + 7) = 184$

(c) (10 Punkte) Es wird sequentiell auf die folgende Adressen (hexadezimal) zugegriffen: F1, AE, 67, F3, CF. Geben Sie für alle vier Caches den Zustand nach jedem Zugriff an. Tragen Sie dazu in die vorgedruckten Tabellen die Adressen der im Cache gespeicherten Daten ein. Zu Beginn sind alle Caches leer.

Lösungsvorschlag:

direct-mapped:

	000	001	010	011	100	101	110	111
F1	Fo, F1							
AE	„							AE, AF
67	„			66, 67				„
F3	„	F2, F3		„				„
CF	„	„		„				CE, CF

2-way set associative:

	Set 0		Set 1		Set 2		Set 3	
	00	00	01	01	10	10	11	11
F1	Fo, F1							
AE	„						AE, AF	
67							„	66, 67
F3			F2, F3				„	„
CF			„				CE, CF	„

4-way set associative:

	Set 0				Set 1			
	0	0	0	0	1	1	1	1
F1	Fo, F1							
AE	„				AE, AF			
67	„				„	66,67		
F3	„				„	„	F2, F3	
CF	„				„	„	„	CE, CF

fully associative:

F1	Fo, F1							
AE	„	AE, AF						
67	„	„	66, 67					
F3	„	„	„	F2, F3				
CF	„	„	„	„	CE, CF			