Aufgabe 1 : Assembler (maximal 21 Punkte)

1.1: Verständnisfragen (maximal 4 Punkte)

(a) (1 Punkt) Kreuzen Sie an, welche der folgenden Statements in MIPS-Assemblercode sich auf die Grösse des erzeugten Maschinenprogramms auswirken:

☒ `.word 0x12345678`

☐ `loop:`

☒ `jr $ra`

☒ `nop`

(b) (1 Punkt) Was versteht man unter einem Instruktionssatz (=Befehlssatz)?
(Antwort in 1 – 2 Sätzen)

Lösungsvorschlag: Ein Befehlssatz ist die Menge von Maschinenbefehlen (Instruktionen), die ein bestimmter Prozessor verarbeiten kann.

□

(c) (2 Punkte) Ein Prozessor ohne Branch-Delay-Slot führt folgenden Code aus:

```
loop: j loop
```

Kann er unter Umständen weitere Befehle abarbeiten? Begründen Sie.

Lösungsvorschlag: Der Prozessor führt eine Endlosschleife aus, die er auf normalem Wege nicht verlassen kann. Dies hindert ihn jedoch nicht daran, auf Interrupts zu reagieren und die entsprechenden Unterbrechungsroutinen auszuführen. Auf diese Weise können z. B. moderne Betriebssysteme stabil weiterlaufen, selbst wenn ein einzelnes Programm nicht mehr reagiert.

□

1.2: Codeübersetzung (maximal 10 Punkte)

Hinweis: Auf der letzten Seite des Prüfungsbogens finden Sie eine Übersicht von Assemblerbefehlen. Alle für diese Aufgabe benötigten Befehle sind dort enthalten.

In Tabelle 1 auf der nächsten Seite sind verschiedene Ausschnitte aus C-Code (mit den Datentypen der benutzen Variablen) und MIPS-Assemblercode gegeben. Dabei ist eine Variable vom Typ `int`*xx*`_t` immer ein *xx* Bit breiter Integer mit Vorzeichen, `uint`*xx*`_t` ohne Vorzeichen. `abs` sei ein Präprozessor-Makro, das heisst, für `abs(x)` wird kein Funktionsaufruf erzeugt, sondern Code, der den Betrag von `x` direkt berechnet. Der verwendete MIPS-Prozessor hat *keinen* Branch-Delay-Slot.

Finden Sie 3 Paare aus C- und Assemblercode, die die gleiche Funkionalität implementieren und geben Sie für jede Variable im C-Code ihre Entsprechung im Assemblercode an. Benutzen Sie hierfür Tabelle 2.

Achtung: Nicht jeder Codeausschnitt hat ein Gegenstück.

Tabelle 1: Ausschnitte aus C-Code (links) und Assemblercode (rechts)

1.
```
if(a > 0) b = 1;
else b = 0;
---
a, b: int32_t
```

2.
```
a = b + c;
if(a < b) d = e;
---
a, b, c, d, e: uint32_t
```

3.
```
a = b + c;
---
a, c: uint64_t,
b: uint32_t
```

4.
```
x = abs(x);  // Betrag von x
---
x: int32_t
```

a)
```
    and   $t1, $zero, $zero
    beq   $zero, $s1, L1
    addiu $t1, $t1, 1
L1:
```

b)
```
    slt $v0, $zero, $a0
```

c)
```
    slt $t1, $zero, $t0
    bne $t1, $zero, L1
    sub $t0, $zero, $t0
L1:
```

d)
```
    addu $s0, $t1, $t2
```

e)
```
    addu $v1, $a1, $a2
    sltu $v0, $v1, $a1
    addu $v0, $v0, $a0
```

Tabelle 2: Zuordnung der Codeausschnitte. **Füllen Sie nicht mehr als 3 Zeilen aus!**

C	MIPS	Variablenzuordnung		
1	b	a: `$a0`	b: `$v0`	
2		*kein passender Assemblercode-Ausschnitt*		
3	e	a: `$v0`(high), `$v1`(low)	b: `$a2`	c: `$a0`(high), `$a1`(low)
4	c	x: `$t0`		

1.3: Funktionen (maximal 7 Punkte)

Eine Funktion liefert in bestimmten Fällen falsche Ergebnisse. Der Fehler wurde gefunden und korrigiert. In untenstehender Tabelle ist links der Originalcode und rechts der korrigierte Code gezeigt. Es wurden nur die Zeilen 12 bis 14 eingefügt.

Es gelten die aus der Vorlesung bekannten Konventionen für Funktionsaufrufe.

```
1   .data
2 gaussvals:
3   .float 0.39894, 0.39886, ...
4
5   .text
6   .globl gauss
7 gauss:
8   slt  $v0, $a0, $zero
9   beq  $v0, $zero, argpos
10  sub  $a0, $zero, $a0
11 argpos:
12
13
14
15  sll  $a0, $a0, 2
16  lw   $v0, gaussvals($a0)
17 return:
18  jr   $ra
```

```
1   .data
2 gaussvals:
3   .float 0.39894, 0.39886, ...
4
5   .text
6   .globl gauss
7 gauss:
8   slt  $v0, $a0, $zero
9   beq  $v0, $zero, argpos
10  sub  $a0, $zero, $a0
11 argpos:
12  add  $v0, $zero, $zero
13  slti $s6, $a0, 200
14  beq  $s6, $zero, return
15  sll  $a0, $a0, 2
16  lw   $v0, gaussvals($a0)
17 return:
18  jr   $ra
```

Die drei Punkte („...“) bei `gaussvals` in Zeile 3 stehen für weitere Fließkommazahlen, insgesamt enthält das Datenfeld 200 Werte.

(a) (4 Punkte) Was war der Fehler, der korrigiert wurde?

> **Lösungsvorschlag:** Für zu große Argumente `$a0` wurde ein Wert außerhalb der gültigen Dimensionen des Datenfelds `gaussvals` geladen. Dies wird nun vorher überprüft und stattdessen Null zurückgegeben.

□

Die korrigierte Funktion wird getestet, sie gibt nun die erwarteten Werte zurück. Als sie allerdings in ein größeres, sonst korrektes Programm eingesetzt wird, liefert dieses falsche Ergebnisse.

(b) (3 Punkte) Wo im Code liegt der Fehler? Erläutern Sie, worin genau er besteht und wie man ihn am einfachsten beheben kann.

Lösungsvorschlag: Die neue Implementierung überschreibt das Register `$s6`, ohne es vorher zu sichern und anschließend zurückzuladen. Laut Aufrufkonvention ist dies nicht zulässig, denn die aufrufende Funktion muss sich darauf verlassen können, dass der Wert des Registers unverändert bleibt.

Am einfachsten lässt sich das Problem beheben, indem man ein `$t`-Register benutzt.

□

Aufgabe 2 : Computerarchitektur (maximal 24 Punkte)

2.1: Datenpfad (maximal 8 Punkte)

In dieser Teilaufgabe wird von einem Prozessor mit folgenden Eingenschaften ausgegangen:

- Es werden insgesamt 5 Pipeliningstufen verwendet (IF, ID, EX, MEM, WB).
- Die EX Stufe der Pipeline dauert mit bis zu $t_{EX} = 210\,\text{ns}$ am längsten, während die ID Stufe mit $t_{ID} = 80\,\text{ns}$ am schnellsten abgearbeitet ist.
- Die Signallaufzeiten des Taktsignals vom Taktgenerator zu den einzelnen Registern zwischen den Pipeline-Stufen liegen im Intervall $[t_{d,min}, t_{d,max}] = [6, 18]\,\text{ns}$.
- Die verwendeten Register haben eine Verzögerung von $t_{clkToQ} = 15\,\text{ns}$ zwischen aktiver Taktflanke und dem Anliegen eines gültigen Ausgangssignals. Die Setup- und Hold-Zeiten der Register liegen bei $t_{setup} = 13\,\text{ns}$ respektive $t_{hold} = 10\,\text{ns}$.

In den folgenden Aufgaben geht es darum, die maximale Taktfrequenz dieses Prozessors zu bestimmen.

(a) (4 Punkte) Beschreiben Sie *formal*, welche beiden Bedingungen die Signallaufzeiten und die Taktperiode T_{clk} erfüllen müssen, damit dieser Prozessor korrekt arbeitet.

Lösungsvorschlag:

$$t_{clkToQ} + t_{ID} \geq \underbrace{t_{d,max} - t_{d,min}}_{t_{clkSkew}} + t_{hold} \tag{1}$$

$$T_{clk} \geq t_{clkToQ} + t_{EX} + t_{setup} + t_{clkSkew} \tag{2}$$

□

(b) (2 Punkte) Welche maximale Taktfrequenz $f_{clk,max}$ resultiert aus diesen Bedingungen?

Lösungsvorschlag:

$$T_{clk,min} = t_{clkToQ} + t_{EX} + t_{setup} + t_{clkSkew} = 15 + 210 + 13 + 12 = 250\,\text{ns} \tag{3}$$

$$f_{clk,max} = (T_{clk,min})^{-1} = 1/250\,\text{ns} = 4\,\text{MHz} \tag{4}$$

□

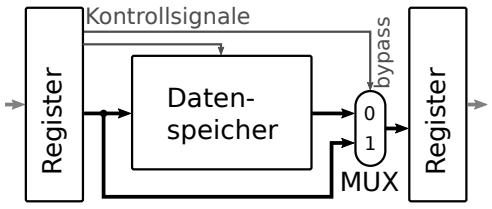

Abbildung 1: Bypass für die MEM Stufe, falls kein Speicherzugriff benötigt wird.

(c) (2 Punkte) Einige Instruktionen (z. B. Verzweigungen) brauchen keinen Speicherzugriff in der MEM Phase. Ist es möglich, das Datenspeicher-Modul wie in Abbildung 1 gezeigt

mit einem Multiplexer zu umgehen? Begründen Sie rechnerisch. Der Multiplexer habe eine Antwortzeit von $t_{mux} = 5$ ns.

Lösungsvorschlag: Erste Ungleichung der Teilaufgabe a) muss erfüllt sein!

$$15 + 5 \not\geq 12 + 10 \Rightarrow \text{nicht erfüllt!} \tag{5}$$

□

2.2: Branch Prediction

(maximal 12 Punkte)

In dieser Aufgabe betrachten wir eine fünfstufige Pipeline-Architektur ohne Cache mit den bekannten Stufen IF, ID, EX, MEM und WB. Die Architektur besitzt eine zusätzliche Logik in den ersten beiden Stufen, um schon frühzeitig in der ID- statt der MEM-Stufe über den Sprung zu entscheiden und die Verzweigungsadresse zu berechnen.

(a) (1 Punkt) Die frühzeitige Entscheidung minimiert die Auswirkungen welches Hazard-Typs?

Lösungsvorschlag: Ablauf Hazard.

□

(b) (1 Punkt) Wie viele Stall-Zyklen müssen ohne beziehungsweise mit frühzeitiger Entscheidung eingefügt werden? Neben den Stall-Zyklen wird keine weitere Technik zur Vermeidung dieser Hazards verwendet.

Lösungsvorschlag:

ohne frühzeitige Entscheidung: ☐ 0 ☐ 1 ☐ 2 ☒ 3 ☐ 4 ☐ 5 Stall-Zyklen

mit frühzeitiger Entscheidung: ☐ 0 ☒ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 Stall-Zyklen

□

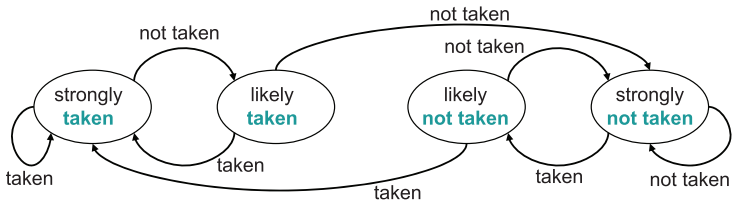

Abbildung 2: 2-Bit-Prädiktion für dynamische Sprungvorhersage.

Für die nachfolgenden Teilaufgaben wird die Architektur zusätzlich mit einem (oder mehreren) 2-Bit-Prädiktor(en) (siehe Abbildung 2) ausgestattet. Der Ausgangszustand des Prädiktors sei dabei immer `likely taken`. Die Verwendung der Prädiktion soll für untenstehendes MIPS Programm analysiert werden, welches die Summennorm eines Vektors (gespeichert als Array) berechnet. Die Startadresse des Arrays wird im Register `$a0` übergeben, die Länge des Arrays im Register `$a1`. Das Ergebnis der Funktion wird im Register `$v0` zurückgegeben, welches zu Beginn mit `0` initialisiert ist.

Betrachten Sie in den Aufgaben jeweils die Ausführung des Programms für folgendes Array mit 5 Elementen: `[ 0, -1, -4, 3, 2 ]`
Verwenden Sie für das Ausfüllen der Tabellen folgende **Abkürzungen für die Zustände**:
`ST` strongly taken, `LT` likely taken, `LNT` likely not taken, `SNT` strongly not taken.

Hinweis: Auf der letzten Seite des Prüfungsbogens finden Sie eine Übersicht von Assemblerbefehlen. Alle in dieser Aufgabe verwendeten Befehle sind dort enthalten.

```
#1    loop:   addi  $a1,$a1,-1
#2            lw    $t0,0($a0)
#3            slt   $t1,$t0,$zero
#4            beq   $t1,$zero,pos
#5            sub   $t0,$zero,$t0
#6    pos:    add   $v0,$v0,$t0
#7            addi  $a0,$a0,4
#8            bne   $a1,$zero,loop
#9            jr    $ra
```

(c) (9 Punkte) Der hier verwendete Prozessor habe nur *einen einzigen* 2-Bit-Prädiktor für alle Entscheidungen. Das heisst, dass **derselbe 2-Bit-Prädiktor für alle Entscheidungen** verwendet wird. Vervollständigen Sie Tabelle 3 mit den Prädiktor-Zuständen, und der Analyse ob die Sprungvorhersage korrekt war.
Verwenden Sie für die Zustände die oben genannten Abkürzungen und ja/nein für die Korrektheit der Sprungvorhersage!

Lösungsvorschlag: Siehe Tabelle 3.

Tabelle 3: Dynamische Sprungvorhersage für einen globalen 2-Bit-Prädiktor.

Iteration	**1**	**2**	**3**	**4**	**5**
Zustand vor `beq` *(#4)*	LT	ST	ST	ST	ST
Sprungvorhersage `beq` *(#4) korrekt*	ja	nein	nein	ja	ja
Zustand vor `bne` *(#8)*	ST	LT	LT	ST	ST
Sprungvorhersage `beq` *(#8) korrekt*	ja	ja	ja	ja	nein

□

(d) (1 Punkt) Macht es Sinn, statt einen globalen 2-Bit-Prädiktor für alle Entscheidungen mehrere 2-Bit-Prädiktoren für jede Verzweigungsinstruktion zu benutzen? Begründen Sie in einem Satz und unabhängig von den Resultaten der vorangehenden Teilaufgaben!

Lösungsvorschlag: Das macht Sinn, z.B. für Situationen, in welchen die Sprungentscheidung innerhalb einer Schleife in den meisten Fällen nicht der Entscheidung der Schleifenbedingung (immer Sprung vorhersagen) entspricht.

□

2.3: Pipelining & Parallelität (maximal 4 Punkte)

(a) (4 Punkte) Treffen folgende Aussagen zu? Kreuzen Sie die jeweils richtige Antwort an. Ein richtig gesetztes Kreuz gibt es 0.5 Punkte, für ein falsch gesetztes Kreuz werden 0.5 Punkte abgezogen. Ein negatives Punktetotal ergibt 0 Punkte.

i. Durch geschicktes Umordnen von Instruktionen wird die Forwarding-Unit eines Prozessors für jedes beliebige Programm hinfällig, da sie dann keinen Einfluss mehr auf die Ausführungszeit des Programms hat.

Lösungsvorschlag:
☐ richtig ☒ falsch
Falsch: Mit Function-Reordering können nicht immer alle Datenabhängigkeiten beseitigt werden.

☐

ii. Wenn eine Kombination von zwei Instruktionen in der Pipeline von der Rechenarchitektur nicht unterstützt wird, spricht man von einem strukturellen Hazard.

Lösungsvorschlag:
☒ richtig ☐ falsch
Richtig: Eine durch die Architektur nicht unterstützte Kombination von Instruktionen ist per Definition ein struktureller Hazard.

☐

iii. Für die Ausführung aller Schleifen mit vielen Iterationen gilt: Es macht keinen Unterschied, ob statische Prädiktion oder dynamische 2-Bit-Prädiktion benutzt wird, das heisst beide, Strategien machen die gleiche Anzahl von falschen Vorhersagen.

Lösungsvorschlag:
☐ richtig ☒ falsch
Falsch: Statische Prädiktion kann auch immer falsch liegen, es gibt keine Garantie dass statische Prädiktion immer die bessere Vorhersage benutzt. Zudem ist die Anzahl falscher Vorhersagen des 2-Bit-Prädiktors von dessen Zustand vor der Schleife abhängig.

☐

iv. Forwarding kann die Zeit zwischen dem Laden einer Instruktion aus dem Speicher und deren Verlassen der Pipeline verringern.

Lösungsvorschlag:
☒ richtig ☐ falsch
Richtig: Forwarding wird fürs Beseitigen von Datenabhängigkeiten verwendet und kann dadurch Stalls nach dem Laden der Instruktion verhindern, welche sonst zusätzliche Taktzyklen für die Ausführung einer Instruktion verlangen.

☐

v. Superpipelining beschleunigt die Programmausführung, da damit die Auswirkungen von Hazards auf die Ausführungszeit reduziert werden.

Lösungsvorschlag:
☐ richtig ☒ falsch
Falsch: Hazards haben hier eine noch grössere Auswirkungen auf die Ausführungszeit!

☐

vi. Um die Pipeline aufgrund eines Hazards zu stallen, wird das Taktsignal des Prozessors angehalten.

Lösungsvorschlag:
☐ richtig ☒ falsch
Falsch: Die Stufen müssen für die vorherige Funktion immer noch weiter ausgeführt werden. Nur einzelne Stufen werden angehalten beziehungsweise deren Taktsignal unterdrückt.

☐

vii. Ein VLIW-MIPS-Prozessor nutzt wie ein Standard-MIPS-Prozessor eine gemeinsame Pipeline, ein gemeinsames Registerfeld und denselben Speicher für die parallel ausgeführten Instruktionen.

Lösungsvorschlag:
☒ richtig ☐ falsch
Richtig. Nur die Anzahl Recheneinheiten, bez. Ports für das Registerfeld/den Speicher werden erweitert.

☐

viii. VLIW MIPS beschleunigt die Programmausführung gegenüber einem Standard-MIPS-Prozessor durch dynamische Parallelität.

Lösungsvorschlag:
☐ richtig ☒ falsch
Falsch: VLIW ist ein Beispiel für *statische* Parallelität.

☐

Aufgabe 3 : Cache (maximal 19 Punkte)

3.1: Cache Grundlagen (maximal 6 Punkte)

(a) (1 Punkt) Erklären Sie die beiden Arten der Lokalität bei Speicherabfragen mit je einem Satz.

Lösungsvorschlag: **Zeitliche Lokalität:** Gleiche Adresse wird innerhalb kurzer Zeit mehrfach referenziert. (0.5P)
Örtliche Lokalität: Beieinander gelegene Adressen werden innerhalb kurzer Zeit referenziert. (0.5P)

□

(b) (1 Punkt) Beschreiben Sie ein kurzes Programm (in Pseudocode), das einen hohen Grad an örtlicher Lokalität aufweist. Begründen Sie ihre Antwort in 1-2 Sätzen.

Lösungsvorschlag: Viele Antworten möglich, typisches Beispiel ist der wiederholte Array-Zugriff. (1P)

□

(c) (1 Punkt) Erklären Sie in 1-2 Sätzen den Unterschied zwischen einem write-through und einem write-back Cache.

Lösungsvorschlag: **Write-through:** Bei jedem Schreibvorgang wird sowohl in den Cache als auch in den Hauptspeicher geschrieben. (0.5P) **Write-back:** Der Prozessor schreibt nur in den Cache. In den Hauptspeicher wird nur geschrieben, wenn ein Block ersetz wird. (0.5P)

□

(d) (1 Punkt) Gegeben ist eine Prozessor-Architektur P1 mit einem L1-Cache (First Level Cache) und einem Hauptspeicher. Nehmen Sie an, T_{L1} sei die Zugriffszeit auf Daten im Cache, H_{L1} die Hit-Rate (relativer Anteil erfolgreicher Zugriffe in L1) und T_{MM} die Miss-Strafe bei einem Hauptspeicher-Zugriff.

Geben Sie die durchschnittliche Zugriffszeit T_{P1} einer Lese-Operation (Read) auf ein Speicherdatum für die Architektur P1 an.

Lösungsvorschlag:

$$T_{P1} = T_{L1} + (1 - H_{L1}) \cdot T_{MM}$$

(1P)

□

(e) (2 Punkte) Gegeben ist eine Prozessor-Architektur P2 mit je einem L1-Cache und-L2 Cache und einem Hauptspeicher. Nehmen Sie an, T_{L1} und T_{L2} seien die Zugriffszeiten auf Daten im L1- bzw. L2-Cache, H_{L1} und H_{L2} die Hit-Raten für den L1- bzw. L2-Cache und T_{MM} die Miss-Strafe bei einem Hauptspeicher-Zugriff.

Geben Sie die durchschnittliche Zugriffszeit T_{P2} einer Lese-Operation (Read) auf ein Speicherdatum für die Architektur P2 an.

Lösungsvorschlag:

$$T_{P2} = T_{L1} + (1 - H_{L1}) \cdot (T_{L2} + (1 - H_{L2}) \cdot T_{MM})$$

(2P)

□

3.2: Ersetzungsstrategie (maximal 13 Punkte)

Eine häufig angewendete Ersetzungsstrategie bei einem Cache-Miss ist das Ersetzen des jeweiligen Blocks, der am längsten nicht referenziert wurde, auch *Least Recently Used* (LRU) genannt.

Eine weitere Strategie, die in modernen Prozessor-Architekturen verwendet wird, nennt sich *Pseudo Least Recently Used* (PLRU). Der Ablauf von PLRU für einen 4-fach assoziativen Cache mit entsprechend 4 Cacheblöcken (L0, L1, L2 und L3) pro Cachezeile und drei Status-Bits (B0, B1 und B2) pro Cachezeile ist in Abbildung 3 abgebildet. Bei einem Cache-Miss wird anhand von B0 entschieden, ob in der Gruppe {L0, L1} oder {L2, L3} ein Block ersetzt wird. Im nächsten Schritt wird der zu ersetzende Block anhand von B1 oder B2 gefunden.

Beim Zugriff auf einen vorhandenen Block werden die Bits, welche im Enscheidungsdiagramm auf dem Weg zum entsprechenden Block liegen, so gesetzt, dass sie auf diejenige Kante zeigen, die nicht zum zugegriffenen Block führt. Zum Beispiel werden nach dem Zugriff auf Block L0 die beiden Bits B0 und B1 auf 1 gesetzt. Der Wert vom B2 wird nicht beeinflusst.

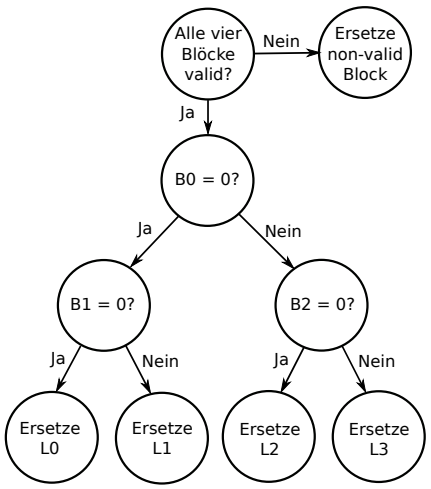

Abbildung 3: Ablauf der PLRU Ersetzungsstrategie für einen 4-fach assoziativen Cache

(a) (3 Punkte) Wie sind die Bits B0, B1 und B2 nach einem Zugriff auf einen der vier Blöcke jeweils gesetzt? Füllen Sie dafür Tabelle 5 aus. Falls ein Bit nicht beeinflusst wird, kennzeichnen Sie dies mit einem *'X'*.

Block-Zugriff	B0	B1	B2
L0	1	1	X
L1			
L2			
L3			

Tabelle 4: Füllen Sie die Tabelle aus.

Lösungsvorschlag:

Block-Zugriff	B0	B1	B2
L0	1	1	X
L1	1	0	X
L2	0	X	1
L3	0	X	0

Tabelle 5: Musterlösung (1P pro Zeile)

□

(b) (3 Punkte) Nennen Sie ein Beispiel einer Block-Zugriffs-Abfolge, an der man sieht, dass sich PLRU nicht wie LRU verhält.

Lösungsvorschlag: Bei der Zugriffs-Abfolge L1 → L2 → L3 → L0 wird im nächsten Schritt Block L2 ersetzt und nicht L1 wie bei LRU. (3P)

□

(c) (6 Punkte) Geben Sie die Anzahl der benötigten Status-Bits für einen allgemeinen N-fach assoziativen Cache mit PLRU-Ersetzungsstrategie an. Wieviele Bits würde man für einen N-fach assoziativen Cache mit einer korrekten Implementierung von LRU mindestens brauchen?

Lösungsvorschlag: LRU: (3P) Bei N Blöcken haben wir $N!$ verschiedene Reihenfolgen der Block-Zugriffe. Um diese zu Codieren benötigt man mindestens

$$\#Bits_{Cache} = \lceil log_2(N!) \rceil$$

PLRU: (3P) Die Anzahl benötigter Bits entspricht der Anzahl Knoten im Entscheidungs-Baum wie in Abbildung 3, die keine Blätter sind. Somit haben wir

$$\#Bits_{Cache} = N - 1$$

Eine weitere Herleitung: Der einfachste Fall ist ein 2-fach assoziativer Cache, für den man $N - 1 = 1$ Status-Bit braucht. Für jeden Cache-Block, um den man den Cache erweitert, benötigt man nun ein zuätzliches Status-Bit.

□

(d) (1 Punkt) Nennen Sie einen Vorteil von PLRU gegenüber LRU.

Lösungsvorschlag: LRU erfordert mehr Status-Bits (1P), d.h. komplexere Hardware, da $\lceil log_2(N!) \rceil > N - 1$ für $N > 2$

□

Anhang

MIPS-Instruktionssatz

Zusammenfassung einiger MIPS-Assemblerinstruktionen, die in den Aufgaben 1 und 2 benötigt werden könnten.

Instruktion	Bedeutung
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`addi $s1, $s2, 100`	`$s1 = $s2 + 100`
`addiu $s1, $s2, 100`	`$s1 = $s2 + 100` (unsigned)
`addu $s1, $s2, $s3`	`$s1 = $s2 + $s3` (unsigned)
`and $s1, $s2, $s3`	`$s1 = $s2 & $s3` (bitweises Und)
`andi $s1, $s2, 100`	`$s1 = $s2 & 100` (bitweises Und)
`beq $s1, $s2, 25`	Wenn `($s1 == $s2)`, verzweige zu `PC + 4 + 100`
`bne $s1, $s2, 25`	Wenn `($s1 != $s2)`, verzweige zu `PC + 4 + 100`
`j label`	Springe zu `label`
`jal label`	Springe zu `label` und setze `$ra`
`jr $s1`	Springe zu `$s1`
`li $s1, 100`	`$s1 = 100`
`lui $s1, 100`	`$s1 = 100 * 2`^{16}
`lw $s1, 100($s2)`	`$s1 = Speicher[$s2 + 100]`
`move $s1, $s2`	`$s1 = $s2`
`mul $s1, $s2, $s3`	`$s1 = $s2 * $s3`
`nop`	No operation
`sll $s1, $s2, 10`	`$s1 = $s2 << 10`
`slt $s1, $s2, $s3`	Setze `$s1=1` falls `($s2 < $s3)`; sonst `$s1=0`
`slti $s1, $s2, 100`	Setze `$s1=1` falls `($s2 < 100)`; sonst `$s1=0`
`sltu $s1, $s2, $s3`	Setze `$s1=1` falls `($s2 < $s3)` (unsigned); sonst `$s1=0`
`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`
`sw $s1, 100($s2)`	`Speicher[$s2 + 100] = $s1`