Aufgabe 1 : Assembler (maximal 27 Punkte)

Der grösste gemeinsame Teiler (GGT) von ganzen Zahlen, die nicht alle gleich null sind, entspricht der grössten natürlichen Zahl, durch die sich diese Zahlen ohne Rest teilen lassen. In dieser Aufgabe befassen wir uns nur mit dem GGT von zwei natürlichen Zahlen. Als Beispiel: Der GGT von *0* und *8* ist *8*; der GGT von *4* und *15* ist *1*; der GGT von *42* und *30* ist *6*.

In der Mathematik ist der euklidische Algorithmus eine effiziente Methode zur Berechnung des GGTs von zwei natürlichen Zahlen. Der euklidische Algorithmus basiert auf dem Prinzip, dass sich der GGT von zwei Zahlen nicht ändert, wenn die grössere Zahl durch ihre Differenz mit der kleineren Zahl ersetzt wird.

Diese Aufgabe ist in **drei** Teile gegliedert, in welchen jeweils ein Algorithmus zur Berechnung des GGTs in MIPS Assembler und/oder als C-Code implementiert wird. Der verwendete MIPS-Prozessor hat **keinen** Branch-Delay-Slot. Der Datentyp **unsigned int** bezeichnet einen vorzeichenlosen Integer-Datentyp mit 4 Bytes.

1.1: Verständnisfragen (maximal 15 Punkte)

Im ersten Teil ist die Originalversion des euklidischen Algorithmus gegeben, in welcher nur Subtraktionen benutzt werden. Gegeben sind die Implementation des Algorithmus sowohl in Assembler auf Seite 2 als auch C auf Seite 3. Beantworten Sie die nachfolgenden Verständnisfragen anhand der gegebenen Code-Blöcken.

Hinweis: Auf der letzten Seite des Prüfungsbogens finden Sie eine Übersicht von Assemblerbefehlen.

Assembler-Code:

```
1      .data
2          na: .word 40
3          nb: .word 24
4      .text
5          .global main
6      main:
7          lw $a0, na($0)
8          lw $a1, nb($0)
9          jal GGT1
10     inf_loop:                        # Das Program endet in
11         j inf_loop                   # einer unendlichen Schleife
12     GGT1:
13         bne $a0, $0, GGT1_loop
14         move $v0, $a1
15         jr $ra
16     GGT1_loop:
17         beq $a1, $0, GGT1_end
18         slt $t4, $a1, $a0
19         beq $t4, $0, GGT1_else
20         sub $a0, $a0, $a1
21         j GGT1_loop
22     GGT1_else:
23         sub $a1, $a1, $a0
24         j GGT1_loop
25     GGT1_end:
26         move $v0, $a0
27         jr $ra
```

C-Code:

```
1  #include <stdio.h>
2  unsigned int a = 40;
3  unsigned int b = 24;
4  unsigned int GGT1(unsigned int x, unsigned int y)
5  {
6      if (x == 0)
7          return y;
8      while (y!=0)
9      {
10         if (x > y)
11             x = x - y;
12         else
13             y = y - x;
14     }
15     return x;
16 }
17 void main ()
18 {
19     unsigned int c = GGT1(a, b);
20     while (true);
21 }
```

(a) (6 Punkte) Geben Sie in Tabelle 1 für die gegebenen Variablen im C-Code ihre Entsprechung im Assembler-Code an.

Tabelle 1: Zuordnung der Variable.

Variable in C	Register in MIPS
c	
x	
y	

(b) (1 Punkt) Die MIPS-Assembler Instruktion `move` ist eine Pseudoinstruktion. Nennen Sie eine Möglichkeit wie die Instruktion in Zeile 14 (Assembler-Code) in eine äquivalente Assembler Instruktion übersetzt werden kann.

```
move $v0, $a1
```

(c) (6 Punkte) Im Hauptspeicher wird die "big-endian" Konvention angewendet. Kodieren Sie die Instruktion in Zeile 23 (Assembler-Code) in **hexadezimaler** Darstellung.
Hinweis: Die Funktionskodierung der Operation `sub` *ist 34; die Registernummer von* `$a0` *ist 4; die Registernummer von* `$a1` *ist 5.*

Geben Sie zudem die **binäre** Darstellung der Instruktionskodierung im Programmspeicher (Tabelle 2) an. Die Adresse der Instruktion ist in Tabelle 2 gegeben.

```
sub $a1, $a1, $a0
```

Hexadezimal Darstellung: 0x____________________

Tabelle 2: Programmspeicher

	Adresse	Inhalt
	...	
	0x00400008	
	0x00400007	
	0x00400006	
	0x00400005	
Adresse der Instruktion `sub $a1, $a1, $a0` ⟶	0x00400004	
	0x00400003	
	0x00400002	
	0x00400001	
	0x00400000	
	...	

(d) (2 Punkte) Ergänzen Sie die Tabelle 3 mit den aus der Vorlesung bekannten Adressierungsarten der Instruktionen.

Tabelle 3: Zuordnung der Adressierungsarten.

Instruktion	Adressierungsart
`lw $a0, na($0)`	
`bne $a0, $0, GGT1_loop`	
`jr $ra`	
`slt $t4, $a1, $a0`	

1.2: Code Übersetzung

(maximal 12 Punkte)

Die Originalversion des euklidischen Algorithmus berechnet eine einzige Subtraktion pro Schleife. Deshalb führt er unnötig viele Schleifen bis zum Abbruch aus, falls ein grosser Unterschied zwischen den beiden natürlichen Zahlen besteht, z.B. bei *1* und *1000*. In einer zweiten Version wird die Subtraktion durch eine Modulo-Operation (Division mit Rest) ersetzt. Der euklidische Algorithmus mit Modulo ist unterhalb der Aufgabe als MIPS Assembler-Code gegeben.

Dieser Assembler-Code ersetzt die Zeilen 12 bis 27 des vorherigen Assembler-Codes auf Seite 2 (Aufgabe 1.1). Die Zeile 9 des vorherigen Assembler-Codes wird zu `jal GGT2` umgeschrieben. Danach kann der gesamte euklidische Algorithmus als Assembler-Code mit Modulo-Operationen realisiert werden.

```
1   GGT2:
2       move $t4, $0            # Hinweis: Definition einer lokalen Variable
3                               # tmp und Initialisierung als 0
4   GGT2_loop:
5       beq $a1, $0, GGT2_end
6       move $t4, $a1
7       div $a0, $a1            # Division: $a0 wird geteilt durch $a1
8                               # Quotient in $LO; Rest in $HI
9       mfhi $a1                # Speichere Rest ($HI) in $a1
10      move $a0, $t4
11      j GGT2_loop
12  GGT2_end:
13      move $v0, $a0
14      jr $ra
```

(a) (6 Punkte) Übersetzen Sie die gegebene Assembler-Implementierung `GGT2` auf Seite 6 in eine semantisch äquivalente C-Funktion `GGT2`, welcher `GGT1` in Aufgabe 1.1 entspricht.

Hinweis: Alle genutzten Variablen werden als **unsigned int** *definiert. Die Modulo-Operation wird in C als % geschrieben.*

```
unsigned int GGT2(unsigned int x, unsigned int y)
{

}
```

Der euklidische Algorithmus kann ebenfalls mittels Rekursion umgesetzt werden. Die rekursive Version GGT3 wird durch den folgenden C-Code beschrieben. Dieser C-Code ersetzt die C-Funktion GGT1 in Aufgabe 1.1.

```
1    unsigned int GGT3(unsigned int x, unsigned int y)
2    {
3        if(y!=0)
4            x = GGT3(y, x%y);
5        return x;
6    }
```

(b) (6 Punkte) Ergänzen Sie den Assembler-Code der Funktion `unsigned int GGT3` auf der nächsten Seite, indem Sie an den markierten Stellen Instruktionen oder Operanden einer Instruktion einfügen.

Der ergänzte Assembler-Code soll die Zeilen 12 bis 27 der ersten Version des Assembler-Codes (Aufgabe 1.1) ersetzen. Die Zeile 9 im früheren Assembler-Code wird zu `jal GGT3` umgeschrieben. Danach sollte der Assembler-Code einen rekursiven euklidischen Algorithmus realisieren.

Benutzen Sie ausschliesslich die Register `$a0`, `$a1`, `$v0`, `$sp`, `$ra` und `$0`, die Labels `GGT3` und `GGT3_if`, und die Instruktionen aus dem Anhang. Verzichten Sie auf die Verwendung des Framepointers `$fp`.

```
1   GGT3:
2       bne ______________________________
3       __________________________________
4       jr $ra
5   GGT3_if:
6       ______________________________
7       __________________________________
8       div $a0, $a1
9       move $a0, $a1
10      mfhi $a1
11      jal ______________________________
12      __________________________________
13      __________________________________
14      __________________________________
```

Aufgabe 2 : Computerarchitektur

(maximal 34 Punkte)

2.1: Richtig oder Falsch?

(maximal 3 Punkte)

Treffen die folgenden Aussagen zu? Kreuzen Sie die richtigen Antworten an. Eine korrekte Antwort gibt 0.5 Punkte. Für eine falsche Antwort werden 0.5 Punkte abgezogen. Die Aufgabe gibt keine negative Gesamtsumme.

- Ein Programm, das für den MIPS Very Long Instruction Word (VLIW) Prozessor kompiliert wurde, benötigt immer mehr Speicherplatz im Instruktions-Speicher als ein Programm, das für einen MIPS Prozessor ohne VLIW kompiliert wurde.
  ☐ richtig ☐ falsch

- Ein Datenhazard in einer 5-stufigen MIPS Pipeline kann entstehen, falls eine Instruktion lesend auf ein Register zugreift, nachdem die direkt davor ausgeführte Instruktion auf das selbe Register schreibend zugegriffen hat.
  ☐ richtig ☐ falsch

- Der Speed-up einer MIPS Architekur mit Pipelining gegenüber der äquivalenten Einzeltakt-Implementierung ist nie grösser als die Anzahl der Pipeline-Stufen.
  ☐ richtig ☐ falsch

- VLIW MIPS beschleunigt die Programmausführung gegenüber einem Standard-MIPS-Prozessor durch statische Parallelität.
  ☐ richtig ☐ falsch

- Um eine Blase in eine Pipeline aufgrund eines Hazards einzufügen, wird das Taktsignal des Prozessors angehalten.
  ☐ richtig ☐ falsch

- Superpipelining beschleunigt die Programmausführung, da damit die Auswirkungen von Hazards auf die Ausführungszeit reduziert werden.
  ☐ richtig ☐ falsch

2.2: MIPS Jump Instruktion

(maximal 5 Punkte)

Die Jump Instruktion `j label` eines 32Bit MIPS Prozessors setzt sich aus der 6-Bit breiten Operations-Kodierung und einem 26-Bit breiten Immediate Anteil zusammen. Die Adresse der Instruktion (`label`), an die gesprungen wird, wird wie folgt berechnet:

1.) Erhöhung des Programm-Counters: `PC = PC + 4`

2.) Logischer Links-Shift des Immediate Anteils um 2Bits: `Immediate = Immediate « 2`

3.) Sign extension des Immediate Anteils von 28 auf 32 Bit

4.) Zusammensetzung der Zieladresse mit den obersten 4-Bits des PC-Registers und dem Immediate Anteil: `PC = (PC & 0xF0000000) | (Immediate & 0x0FFFFFFF)`
(`&` = logisches Und, `|` = logisches Oder)

(a) (1 Punkt) Wie nennt sich die Adressierungsart der Jump (`j label`) Instruktion?

(b) (1 Punkt) Wieso wird der Immediate Anteil in Schritt 2.) logisch nach links geschoben?

(c) (2 Punkte) Nehmen Sie nun an, an der Speicher-Adresse `0x0040F300` befindet sich eine jump-Instruktion eines Programms. Welche Adressen können die Instruktionen haben, an die mit einer solchen jump-Instruktion gesprungen werden kann?

(d) (1 Punkt) Nennen Sie eine Möglichkeit, wie man zu einer beliebigen Adresse springen könnte?

2.3: Daten- und Kontrollpfad

(maximal 11 Punkte)

Gegeben ist die Einzeltakt-Implementierung der in der Vorlesung behandelten MIPS Prozessor-Architektur, abgebildet in Abb. 1.

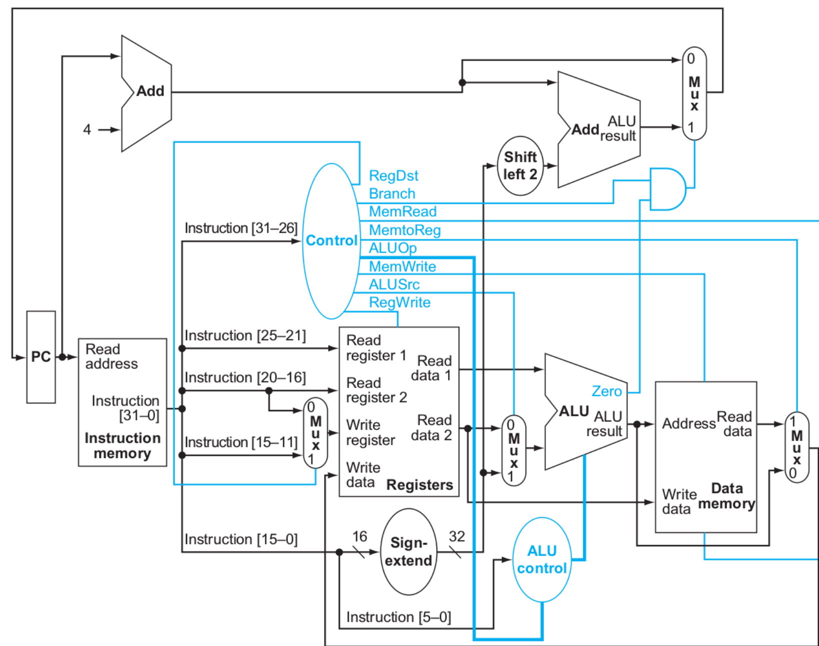

Abbildung 1: Daten- und Kontrollpfad (blaue Verbindungen und Blöcke) einer wie in der Vorlesung behandelten Einzeltaktimplementierung.

Die Architektur hat zudem die folgenden Eigenschaften:

- Es können die folgenden Instruktionen ausgeführt werden: Arithmetische und logische Instruktionen als I oder R Kodierung, z.B. `add, addi, sub, subi, and, andi` etc., Speicher-Referenz-Instruktionen `lw` und `sw` sowie die Verzweigungs-Instruktion `beq`.
- Die Latenzen der einzelnen logischen Blöcke der Architektur sind in Tabelle 4 gegeben.
- Die Kontroll-Einheit (Control) steuert die folgenden 8 Signale des Kontroll-Pfades: *RegDest, Branch, MemRead, MemToReg, AluOp, MemWrite, AluSrc, RegWrite*.

Hinweis: Es bestehen Abhängigkeiten zwischen den Aufgaben. Falls Sie Teilaufgabe a) nicht lösen konnten, verwenden Sie Variabeln für allenfalls benötigte Terme in Teilaufgabe b).

Logischer Block	Latenz
PC (PC-Register) lesen (ClkToQ) oder schreiben (Setup Zeit)	40 ps
Instruction memory (Instruktion-Speicher) lesen	400 ps
Add (Addierer)	50 ps
ALU (Arithmetisch-logische Einheit)	200 ps
ALU control (ALU Kontroll-Einheit)	50 ps
Mux (Multiplexer)	5 ps
Registers (Register-Feld) lesen	150 ps
Registers (Register-Feld) schreiben*	100 ps
Data memory (Daten-Speicher) lesen	500 ps
Data memory (Daten-Speicher) schreiben*	0 ps
Sign extend (Vorzeichenerweiterung)	15 ps
Shift left 2 (Logischer Linksshift)	15 ps
AND Gatter	0 ps

Tabelle 4: Latenzen der einzelnen Blöcke des Prozessors in Abb. 1
**Die Schreib-Latenz des Register-Felds und des Daten-Speichers bezieht sich auf die minimal benötigte Zeit zwischen den jeweiligen Eingangssignalen und der Taktflanke.*

(a) (7 Punkte) Nehmen Sie an, die Kontroll-Einheit (Control) verursacht keine Latenzen zur Generierung der Signale. Berechnen Sie die minimal zulässige Taktperiode der gegebenen Architektur.

(b) (4 Punkte) Nehmen Sie nun an, dass die Kontroll-Einheit auch Latenzen zur Generierung der Kontroll-Signale verursacht. Wie lange darf die Kontroll-Einheit maximal zur Generierung des Kontroll-Signales *MemWrite* brauchen, ohne dabei die minimal zulässige Taktperiode zu beeinflussen?

2.4: Pipelining

(maximal 15 Punkte)

In dieser Aufgabe betrachten wir den in der Vorlesung behandelten MIPS Prozessor mit 5-Stufiger Pipeline (IF → ID → EX → MEM → WB). Die Architektur besitzt folgende Eigenschaften:

- Unterstütztes Forwarding: EX → EX und MEM → EX.
- Alle Sprungentscheidungen werden perfekt vorausgesagt, d.h. nach einer Branch-Instruktion wird immer die korrekte Instruktion in die Pipeline geladen.
- Es wird kein Branch-Delay-Slot verwendet.
- Taktperiode: 100 ns

Gegeben ist folgende C-Funktion `foo`, wobei `unsigned int n` die Länge des Arrays `arr` ist und $n > 0$ gilt:

```
1 int foo(int *arr, unsigned int n){
2     int i = 0;
3     for (i=0;i<n;i++){
4         arr[i] += i*arr[i];
5     }
6 }
```

Für den obigen Prozessor ist die folgende Umsetzung in Assembler gegeben. Dabei gilt beim Funktionsaufruf Register `$a0` = erste Adresse des Arrays `int* arr` und Register `$a1` = `unsigned int n`:

```
1 foo:    move   $t0, $zero
2 loop:   addi   $t0, $t0, 1
3         lw     $t1, 0($a0)
4         mul    $t1, $t1, $t0
5         sw     $t1, 0($a0)
6         addi   $a0, $a0, 4
7         bne    $t0, $a1, loop
8         jr     $ra
```

Hinweis: Auf der letzten Seite des Prüfungsbogens finden Sie eine Übersicht von Assemblerbefehlen.

(a) (4 Punkte) In Tabelle 5 ist die zeitliche Belegung der Pipeline für eine Schleifenausführung gegeben. Zeichnen Sie alle Situationen ein, in denen Forwarding verwendet wird. Verwenden Sie dazu Pfeile, d.h. ↓: Pipeline Phase, die ein Argument forwarded, ↑: Pipeline-Phase, die ein Argument via Forwarding erhält.

	1	2	3	4	5	6	7	8	9	10	11	12
move \$t0, \$zero	IF	ID	EX	MEM	WB							
addi \$t0, \$t0, 1		IF	ID	EX	MEM	WB						
lw \$t1, 0(\$a0)			IF	ID	EX	MEM	WB					
mul \$t1, \$t1, \$t0				IF	ID	-	EX	MEM	WB			
sw \$t1, 0(\$a0)					IF	-	ID	EX	MEM	WB		
addi \$a0, \$a0, 4							IF	ID	EX	MEM	WB	
bne \$t0, \$a1, loop								IF	ID	EX	MEM	WB

Tabelle 5: Zeitliche Belegung der Pipeline für eine Schleifenausführung

(b) (1 Punkt) Im 6. Zyklus muss die Pipeline aufgrund eines Hazards gestallt werden. Um welchen Hazard handelt es sich dabei?

(c) (2 Punkte) Nehmen Sie nun an $n = 4$. Wie lange braucht die Architektur um das gegebene Assembler Programm auszuführen?

(d) (8 Punkte) Nehmen Sie nun an, dass n immer ein ganzzahliges Vielfaches von 2 ist. Optimieren Sie die Ausführungszeit des Programms, indem Sie loop-unrolling mit einem Entfaltungsfaktor 2 anwenden. Es steht Ihnen dabei ein zusätzliches Register `$t2` zur Verfügung. Berechnen Sie wiederum die Ausführungszeit des durch loop-unrolling optimierten Programms in Abhängigkeit zu n.

Aufgabe 3 : Cache und Virtual Memory (maximal 29 Punkte)

3.1: Grundlagen (maximal 3 Punkte)

Treffen die folgenden Aussagen für die in der Vorlesung behandelten Speicherarchitekturen zu? Kreuzen Sie die richtigen Antworten an. Eine korrekte Antwort gibt 0.5 Punkte. Für eine falsche Antwort werden 0.5 Punkte abgezogen. Die Aufgabe gibt keine negative Gesamtsumme.

- Ein Cache Treffer (hit) erfolgt immer bei übereinstimmenden Tags.
  ☐ richtig ☐ falsch
- Aufeinanderfolgendes Schreiben innerhalb eines Cache-Blockes führt bei einem auf Zurückkopieren ("write back") basierendem Cache dazu, dass dieser Cache-Block mehrmals hintereinander in den Hauptspeicher kopiert wird.
  ☐ richtig ☐ falsch
- Snooping Protokolle werden implementiert um Cache-Kohärenz zu ermöglichen.
  ☐ richtig ☐ falsch
- Der Adressbereich des virtuellen Speichers kann grösser sein als der Adressbereich des verfügbaren Hauptspeichers.
  ☐ richtig ☐ falsch
- Der Seitenoffset eines Datums wird bei der Übersetzung von virtuellen Adressen in physikalische Adressen aus der Seitentabelle gelesen.
  ☐ richtig ☐ falsch
- Ein TLB Miss tritt ein wenn sich die Seite nicht im Hauptspeicher befindet oder wenn sich die Seite im Hauptspeicher befindet, jedoch der Eintrag im TLB fehlt.
  ☐ richtig ☐ falsch

3.2: Cache und Busbreite

(maximal 12 Punkte)

Gegeben ist ein Computer-System mit den folgenden Eigenschaften

1. Für Instruktionen und Daten werden getrennte Caches genutzt.
2. Die Speicher- und Busbreite beträgt 1 Wort.
3. Jeder Cache Block ist zwei Wörter lang, der gesamte Block wird bei einem Cache Miss aus dem Hauptspeicher gelesen. Bei einem Write-Miss wird der Block zuerst in den Daten-Cache geschrieben und dann durch die Write-Operation überschrieben.
4. Der Daten-Cache hat eine Treffer (hit) Rate von 95%.
5. 75% aller Speicherzugriffe auf den Daten-Cache sind Lese-Zugriffe, 25% der Speicherzugriffe auf den Daten-Cache sind Schreib-Zugriffe.
6. Im Daten-Cache sind stets 30% aller Blöcke modifiziert (Dirty-Bit ist gesetzt).

Hinweis: Alle Aufgaben können unabhängig von einander gelöst werden.

(a) (6 Punkte) Berechnen Sie die durchschnittliche Anzahl Hauptspeicherzugriffe pro Daten-Cache Zugriff für einen auf Zurückkopieren ("write back") basierenden Cache.

(b) (4 Punkte) Berechnen Sie die durchschnittliche Anzahl Hauptspeicherzugriffe pro Daten-Cache Zugriff für einen auf durchgängigen Schreiben ("write through") basierenden Cache. Nehmen Sie an, dass bei einem Write nur das veränderte Wort im Hauptspeicher aktualisiert wird und nicht der gesamte Block. Beachten Sie weiterhin Punkt 3 der Beschreibung des Computer-Systems.

Nehmen Sie nun an, dass das System zusätzlich folgende Eigenschaften aufweist

- Für einen perfekten Instruktions-Cache und perfekten Daten-Cache beträgt die mittler Zahl der Taktzyklen pro Instruktion (CPI) 2 Zyklen. Der Instruktions-Cache wird weiterhin als perfekt angenommen.
- 30% der Instruktionen greifen auf den Daten-Cache zu.
- Ein Bustransfer besteht aus einem Zyklus zur Übertragung der Adresse, 10 Zyklen für jeden Hauptspeicherzugriff und einem Zyklus pro Datentransfer.

(c) (2 Punkte) Nehmen Sie nun an, dass die mittlere Anzahl Hauptspeicherzugriffe pro Daten-Cache Zugriff 0.2 ist. Berechnen Sie die mittlere Anzahl Taktzyklen pro Instruktion (CPI).

3.3: Virtual memory und TLB (maximal 14 Punkte)

Gegeben ist ein Computer-System das einen virtuellen Speicher und einen Translation Lookaside Buffer (TLB) verwendet. Das System besitzt keinen Cache.

Die virtuelle Adresse hat eine Länge von 36 Bits und die physikalische Adresse eine Länge von 31 Bits. Die physikalische und virtuelle Seitengrösse beträgt jeweils 4 KByte. Der TLB ist 16-fach assoziativ und hat 64 Einträge.

Nehmen Sie zudem an, dass der virtuelle und physikalische Speicher Byte-adressiert sind.

Hinweis: 1 GByte = 1024 MBytes, 1 MByte = 1024 KBytes, 1 KByte = 1024 Bytes

(a) (1 Punkt) Berechnen Sie die Anzahl Bits, welche für den Page Offset verwendet werden.

(b) (1 Punkt) Berechnen Sie die Anzahl Einträge in der Seitentabelle.

(c) (1 Punkt) Neben der physikalischen Seitennummer beinhaltet die Seitentabelle pro Eintrag auch noch ein Valid-Bit und ein Dirty-Bit. Andere Statusbits (LRU, Zugriffsberechtigung, ect.) sind in dieser Teilaufgabe nicht zu berücksichtigen. Berechnen Sie die gesamte Grösse der Seitentabelle.

(d) (4 Punkte) Wie gross ist ein Eintrag in dem TLB (in Anzahl Bits)? Nehmen Sie dazu an, dass ein Valid-Bit und Dirty-Bit Teil des Eintrags sind. Andere Statusbits (LRU, Zugriffsberechtigung, ect.) sind in dieser Teilaufgabe nicht zu berücksichtigen.

(e) (3 Punkte) Wie viele Stufen sind in einer hierarchischen Seitentabelle notwendig, unter der Voraussetzung, dass alle Seitentabellen in Seiten (Grösse von 4 KByte) passen müssen? Nehmen Sie einfachheitshalber an, dass jeder Tabelleneintrag auf allen hierarchischen Stufen jeweils 4 Bytes beansprucht.

(f) (1 Punkt) Wie viele Hauptspeicherzugriffe sind bei einem TLB Miss notwendig um auf ein Datum zuzugreifen? Nehmen Sie dazu an, dass es sich um eine zweistufige Seitentabelle handelt und dass die Seitentabellen und Seitentabelleneinträge welche referenziert werden, existieren und sich im Hauptspeicher befinden.

Hinweis: Diese Aufgabe ist unabhängig von Teilaufgabe (e).

Um zu bestimmen welche Seite bei einem Seitenfehler ersetzt wird, ist die Approximation des Least Recently Used (LRU) Algorithmus mittels Referenz-Bit (second chance) implementiert. Zu jeder Seite gehört ein Referenz-Bit, welches vereinfacht festhält, wie lange diese Seite nicht mehr referenziert wurde.

Hinweis: Der Zeiger ändert sich nur im Falle eines Seitenfehlers

(g) (3 Punkte) Nehmen Sie an, dass der Hauptspeicher nur 3 physikalische Seiten beinhalten kann. Zu Beginn befindet sich das System im Anfangszustand (abgebildet in der ersten Tabelle). Es wird nun in der abgebildeten Reihenfolge auf folgende Seiten zugegriffen:

1, 4, 2

Füllen Sie eine Tabelle pro Speicherzugriff aus. Markieren Sie die momentane Position des Zeigers und füllen Sie die Werte aller Referenz-Bits und die sich im Speicher befindenden Seiten aus.

Zeiger	Ref. Bit	Seite
	1	0
	1	4
⇒	1	2

Anfangszustand

Zeiger	Ref. Bit	Seite

Nach dem Zugriff auf 1

Zeiger	Ref. Bit	Seite

Nach dem Zugriff auf 4

Zeiger	Ref. Bit	Seite

Nach dem Zugriff auf 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`
`beqz $s1, label`	Wenn `($s1 == 0)`, verzweige zu `label`
`bne $s1, $s2, 25`	Wenn `($s1 != $s2)`, verzweige zu `PC + 4 + 100`
`bnez $s1, label`	Wenn `($s1 != 0)`, verzweige zu `label`
`div $s1, $s2`	Teile `$s1` durch `$s2` und speichere den Quotienten in `$LO` und den Restbetrag in `$HI`
`j label`	Springe zu `label`
`jal label`	Springe zu `label` und setze `$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]`
`mfhi $s1`	`$s1 = $HI`
`mflo $s1`	`$s1 = $LO`
`move $s1, $s2`	`$s1 = $s2` (Pseudoinstruktion)
`mul $s1, $s2, $s3`	`$s1 = $s2 * $s3` (Multiplikation)
`nop`	No operation
`or $s1, $s2, $s3`	`$s1 = $s2 | $s3` (bitweises Oder)
`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`
`srl $s1, $s2, 10`	`$s1 = $s2 >> 10`
`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`
`sb $s1, 100($s2)`	`Speicher[$s2 + 100] = $s1` (ein Byte)
`sw $s1, 100($s2)`	`Speicher[$s2 + 100] = $s1` (ein Wort)

Aufgabe 1 : Assembler (maximal 27 Punkte)

Der grösste gemeinsame Teiler (GGT) von ganzen Zahlen, die nicht alle gleich null sind, entspricht der grössten natürlichen Zahl, durch die sich diese Zahlen ohne Rest teilen lassen. In dieser Aufgabe befassen wir uns nur mit dem GGT von zwei natürlichen Zahlen. Als Beispiel: Der GGT von *0* und *8* ist *8*; der GGT von *4* und *15* ist *1*; der GGT von *42* und *30* ist *6*.

In der Mathematik ist der euklidische Algorithmus eine effiziente Methode zur Berechnung des GGTs von zwei natürlichen Zahlen. Der euklidische Algorithmus basiert auf dem Prinzip, dass sich der GGT von zwei Zahlen nicht ändert, wenn die grössere Zahl durch ihre Differenz mit der kleineren Zahl ersetzt wird.

Diese Aufgabe ist in **drei** Teile gegliedert, in welchen jeweils ein Algorithmus zur Berechnung des GGTs in MIPS Assembler und/oder als C-Code implementiert wird. Der verwendete MIPS-Prozessor hat **keinen** Branch-Delay-Slot. Der Datentyp **unsigned int** bezeichnet einen vorzeichenlosen Integer-Datentyp mit 4 Bytes.

1.1: Verständnisfragen (maximal 15 Punkte)

Im ersten Teil ist die Originalversion des euklidischen Algorithmus gegeben, in welcher nur Subtraktionen benutzt werden. Gegeben sind die Implementation des Algorithmus sowohl in Assembler auf Seite 2 als auch C auf Seite 3. Beantworten Sie die nachfolgenden Verständnisfragen anhand der gegebenen Code-Blöcken.

Hinweis: Auf der letzten Seite des Prüfungsbogens finden Sie eine Übersicht von Assemblerbefehlen.

Assembler-Code:

```
1      .data
2          na: .word 40
3          nb: .word 24
4      .text
5          .global main
6      main:
7          lw $a0, na($0)
8          lw $a1, nb($0)
9          jal GGT1
10     inf_loop:                       # Das Program endet in
11         j inf_loop                  # einer unendlichen Schleife
12     GGT1:
13         bne $a0, $0, GGT1_loop
14         move $v0, $a1
15         jr $ra
16     GGT1_loop:
17         beq $a1, $0, GGT1_end
18         slt $t4, $a1, $a0
19         beq $t4, $0, GGT1_else
20         sub $a0, $a0, $a1
21         j GGT1_loop
22     GGT1_else:
23         sub $a1, $a1, $a0
24         j GGT1_loop
25     GGT1_end:
26         move $v0, $a0
27         jr $ra
```

C-Code:

```
1  #include <stdio.h>
2  unsigned int a = 40;
3  unsigned int b = 24;
4  unsigned int GGT1(unsigned int x, unsigned int y)
5  {
6      if (x == 0)
7          return y;
8      while (y!=0)
9      {
10         if (x > y)
11             x = x - y;
12         else
13             y = y - x;
14     }
15     return x;
16 }
17 void main ()
18 {
19     unsigned int c = GGT1(a, b);
20     while (true);
21 }
```

(a) (6 Punkte) Geben Sie in Tabelle 1 für die gegebenen Variablen im C-Code ihre Entsprechung im Assembler-Code an.

Tabelle 1: Zuordnung der Variable.

Variable in C	Register in MIPS
c	$v0
x	$a0
y	$a1

(b) (1 Punkt) Die MIPS-Assembler Instruktion `move` ist eine Pseudoinstruktion. Nennen Sie eine Möglichkeit wie die Instruktion in Zeile 14 (Assembler-Code) in eine äquivalente Assembler Instruktion übersetzt werden kann.

```
move $v0, $a1
```

Lösungsvorschlag:

`add $v0, $0, $a1` oder
`addi $v0, $a1, 0`

□

(c) (6 Punkte) Im Hauptspeicher wird die "big-endian" Konvention angewendet. Kodieren Sie die Instruktion in Zeile 23 (Assembler-Code) in **hexadezimaler** Darstellung.
Hinweis: Die Funktionskodierung der Operation `sub` *ist 34; die Registernummer von* `$a0` *ist 4; die Registernummer von* `$a1` *ist 5.*

Geben Sie zudem die **binäre** Darstellung der Instruktionskodierung im Programmspeicher (Tabelle 2) an. Die Adresse der Instruktion ist in Tabelle 2 gegeben.

```
sub $a1, $a1, $a0
```

Hexadezimal Darstellung: 0x00a42822

Tabelle 2: Programmspeicher

	Adresse	Inhalt
	...	
	0x00400008	
	0x00400007	0010 0010
	0x00400006	0010 1000
	0x00400005	1010 0100
Adresse der Instruktion `sub $a1, $a1, $a0` ⟶	0x00400004	0000 0000
	0x00400003	
	0x00400002	
	0x00400001	
	0x00400000	
	...	

(d) (2 Punkte) Ergänzen Sie die Tabelle 3 mit den aus der Vorlesung bekannten Adressierungsarten der Instruktionen.

Tabelle 3: Zuordnung der Adressierungsarten.

Instruktion	Adressierungsart
`lw $a0, na($0)`	Basisadressierung
`bne $a0, $0, GGT1_loop`	PC-relative Adressierung
`jr $ra`	Registeradressierung
`slt $t4, $a1, $a0`	Registeradressierung

1.2: Code Übersetzung

(maximal 12 Punkte)

Die Originalversion des euklidischen Algorithmus berechnet eine einzige Subtraktion pro Schleife. Deshalb führt er unnötig viele Schleifen bis zum Abbruch aus, falls ein grosser Unterschied zwischen den beiden natürlichen Zahlen besteht, z.B. bei *1* und *1000*. In einer zweiten Version wird die Subtraktion durch eine Modulo-Operation (Division mit Rest) ersetzt. Der euklidische Algorithmus mit Modulo ist unterhalb der Aufgabe als MIPS Assembler-Code gegeben.

Dieser Assembler-Code ersetzt die Zeilen 12 bis 27 des vorherigen Assembler-Codes auf Seite 2 (Aufgabe 1.1). Die Zeile 9 des vorherigen Assembler-Codes wird zu `jal GGT2` umgeschrieben. Danach kann der gesamte euklidische Algorithmus als Assembler-Code mit Modulo-Operationen realisiert werden.

```
1   GGT2:
2       move $t4, $0              # Hinweis: Definition einer lokalen Variable
3                                 # tmp und Initialisierung als 0
4   GGT2_loop:
5       beq $a1, $0, GGT2_end
6       move $t4, $a1
7       div $a0, $a1              # Division: $a0 wird geteilt durch $a1
8                                 # Quotient in $LO; Rest in $HI
9       mfhi $a1                  # Speichere Rest ($HI) in $a1
10      move $a0, $t4
11      j GGT2_loop
12  GGT2_end:
13      move $v0, $a0
14      jr $ra
```

(a) (6 Punkte) Übersetzen Sie die gegebene Assembler-Implementierung `GGT2` auf Seite 6 in eine semantisch äquivalente C-Funktion `GGT2`, welcher `GGT1` in Aufgabe 1.1 entspricht.
Hinweis: Alle genutzten Variablen werden als **unsigned int** *definiert. Die Modulo-Operation wird in C als % geschrieben.*

```
1      unsigned int GGT2(unsigned int x, unsigned int y)
2      {
3          unsigned int tmp = 0;
4          while (y!=0)
5          {
6              tmp = y;
7              y = x % y;
8              x = tmp;
9          }
10         return x;
11     }
```

Der euklidische Algorithmus kann ebenfalls mittels Rekursion umgesetzt werden. Die rekursive Version `GGT3` wird durch den folgenden C-Code beschrieben. Dieser C-Code ersetzt die C-Funktion `GGT1` in Aufgabe 1.1.

```
1   unsigned int GGT3(unsigned int x, unsigned int y)
2   {
3       if(y!=0)
4           x = GGT3(y, x%y);
5       return x;
6   }
```

(b) (6 Punkte) Ergänzen Sie den Assembler-Code der Funktion `unsigned int GGT3` auf der nächsten Seite, indem Sie an den markierten Stellen Instruktionen oder Operanden einer Instruktion einfügen.

Der ergänzte Assembler-Code soll die Zeilen 12 bis 27 der ersten Version des Assembler-Codes (Aufgabe 1.1) ersetzen. Die Zeile 9 im früheren Assembler-Code wird zu `jal GGT3` umgeschrieben. Danach sollte der Assembler-Code einen rekursiven euklidischen Algorithmus realisieren.

Benutzen Sie ausschliesslich die Register `$a0`, `$a1`, `$v0`, `$sp`, `$ra` und `$0`, die Labels `GGT3` und `GGT3_if`, und die Instruktionen aus dem Anhang. Verzichten Sie auf die Verwendung des Framepointers `$fp`.

```
1   GGT3:
2       bne $a1, $0, GGT3_if
3       move $v0, $a0
4       jr $ra
5   GGT3_if:
6       addi $sp, $sp, -4
7       sw $ra, 0($sp)
8       div $a0, $a1
9       move $a0, $a1
10      mfhi $a1
11      jal GGT3
12      lw $ra, 0($sp)
13      addi $sp, $sp, 4
14      jr $ra
```

Aufgabe 2 : Computerarchitektur (maximal 34 Punkte)

2.1: Richtig oder Falsch? (maximal 3 Punkte)

Treffen die folgenden Aussagen zu? Kreuzen Sie die richtigen Antworten an. Eine korrekte Antwort gibt 0.5 Punkte. Für eine falsche Antwort werden 0.5 Punkte abgezogen. Die Aufgabe gibt keine negative Gesamtsumme.

- Ein Programm, das für den MIPS Very Long Instruction Word (VLIW) Prozessor kompiliert wurde, benötigt immer mehr Speicherplatz im Instruktions-Speicher als ein Programm, das für einen MIPS Prozessor ohne VLIW kompiliert wurde.
  ☐ richtig ☐ falsch

  Lösungsvorschlag: ☐ richtig ☒ falsch
  Falsch. Falls die Parallelität perfekt ausgenutzt wird (CPI = 0.5), benötigen beide Programme gleich viel Speicherplatz.

  ☐

- Ein Datenhazard in einer 5-stufigen MIPS Pipeline kann entstehen, falls eine Instruktion lesend auf ein Register zugreift, nachdem die direkt davor ausgeführte Instruktion auf das selbe Register schreibend zugegriffen hat.
  ☐ richtig ☐ falsch

  Lösungsvorschlag: ☒ richtig ☐ falsch
  Richtig. Das Ergebnis von der schreibenden Instruktion ist noch nicht zurückgeschrieben in der ID Phase der darauffolgenden lesenden Instruktion.

  ☐

- Der Speed-up einer MIPS Architekur mit Pipelining gegenüber der äquivalenten Einzeltakt-Implementierung ist nie grösser als die Anzahl der Pipeline-Stufen.
  ☐ richtig ☐ falsch

  Lösungsvorschlag: ☒ richtig ☐ falsch
  Richtig. Im perfekten Fall wird die Pipeline gefüllt und beendet in jedem Zyklus eine Instruktion. Zusammen mit dem Overhead, den es braucht um die Pipeline zu füllen, konvergiert der Speedup zu den Anzahl Pipeline Stufen.

  ☐

- VLIW MIPS beschleunigt die Programmausführung gegenüber einem Standard-MIPS-Prozessor durch statische Parallelität.
  ☐ richtig ☐ falsch

  Lösungsvorschlag: ☒ richtig ☐ falsch
  Richtig.

  ☐

- Um eine Blase in eine Pipeline aufgrund eines Hazards einzufügen, wird das Taktsignal des Prozessors angehalten.
  ☐ richtig ☐ falsch

  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.

  ☐

- Superpipelining beschleunigt die Programmausführung, da damit die Auswirkungen von Hazards auf die Ausführungszeit reduziert werden.
  ☐ richtig ☐ falsch

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

  ☐

2.2: MIPS Jump Instruktion (maximal 5 Punkte)

Die Jump Instruktion `j label` eines 32Bit MIPS Prozessors setzt sich aus der 6-Bit breiten Operations-Kodierung und einem 26-Bit breiten Immediate Anteil zusammen. Die Adresse der Instruktion (`label`), an die gesprungen wird, wird wie folgt berechnet:

1.) Erhöhung des Programm-Counters: `PC = PC + 4`

2.) Logischer Links-Shift des Immediate Anteils um 2Bits: `Immediate = Immediate « 2`

3.) Sign extension des Immediate Anteils von 28 auf 32 Bit

4.) Zusammensetzung der Zieladresse mit den obersten 4-Bits des PC-Registers und dem Immediate Anteil: `PC = (PC & 0xF0000000) | (Immediate & 0x0FFFFFFF)`
(`&` = logisches Und, `|` = logisches Oder)

(a) (1 Punkt) Wie nennt sich die Adressierungsart der Jump (`j label`) Instruktion?

Lösungsvorschlag: Pseudodirekte Adressierung

□

(b) (1 Punkt) Wieso wird der Immediate Anteil in Schritt 2.) logisch nach links geschoben?

Lösungsvorschlag: Der MIPS Speicher ist Byte-adressiert, die Instruktionen aber 4-Byte (=1 Word) lang. Somit beginnt eine neue Instruktion alle 4 Bytes. Der Bit-Shift nach links um 2 (= x4) sorgt dafür, dass nur Adressen, die ein Vielfaches von 4 sind, verwendet werden.

□

(c) (2 Punkte) Nehmen Sie nun an, an der Speicher-Adresse `0x0040F300` befindet sich eine jump-Instruktion eines Programms. Welche Adressen können die Instruktionen haben, an die mit einer solchen jump-Instruktion gesprungen werden kann?

Lösungsvorschlag: Ein Programm darf nur innerhalb einem von 16 Speicherblöcken à 2^{28} platizert werden. Die Grenzen sind durch die obersten 4-Bits gegeben. Somit lautete der zulässige Adress-Bereich: `[0x00000000, 0x0FFFFFFC]`

□

(d) (1 Punkt) Nennen Sie eine Möglichkeit, wie man zu einer beliebigen Adresse springen könnte?

Lösungsvorschlag: Register-Adressierung, wobei Adresse in einem Register gespeichert wird: `jr $t0`

2.3: Daten- und Kontrollpfad

(maximal 11 Punkte)

Gegeben ist die Einzeltakt-Implementierung der in der Vorlesung behandelten MIPS Prozessor-Architektur, abgebildet in Abb. 1.

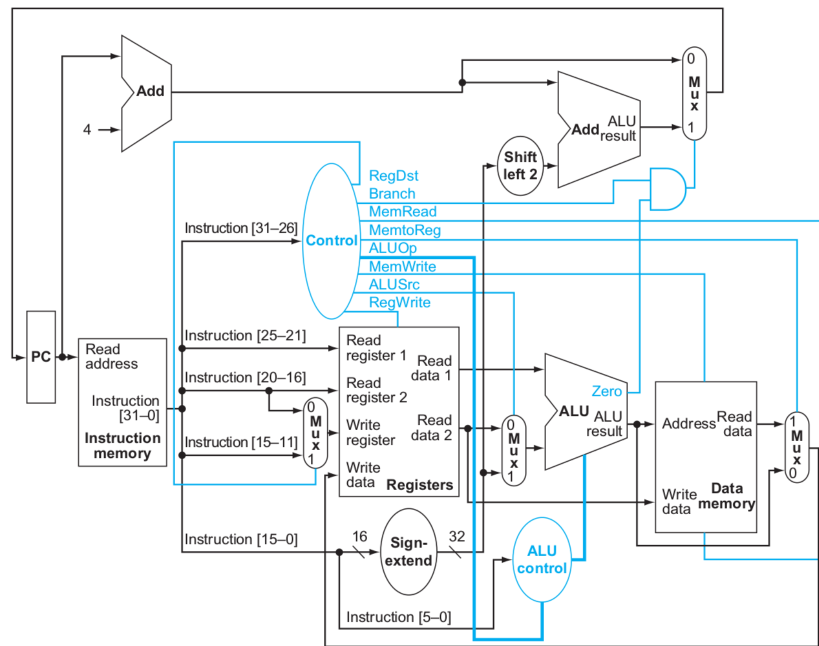

Abbildung 1: Daten- und Kontrollpfad (blaue Verbindungen und Blöcke) einer wie in der Vorlesung behandelten Einzeltaktimplementierung.

Die Architektur hat zudem die folgenden Eigenschaften:

- Es können die folgenden Instruktionen ausgeführt werden: Arithmetische und logische Instruktionen als I oder R Kodierung, z.B. `add, addi, sub, subi, and, andi` etc., Speicher-Referenz-Instruktionen `lw` und `sw` sowie die Verzweigungs-Instruktion `beq`.
- Die Latenzen der einzelnen logischen Blöcke der Architektur sind in Tabelle 4 gegeben.
- Die Kontroll-Einheit (Control) steuert die folgenden 8 Signale des Kontroll-Pfades: *RegDest, Branch, MemRead, MemToReg, AluOp, MemWrite, AluSrc, RegWrite*.

Hinweis: Es bestehen Abhängigkeiten zwischen den Aufgaben. Falls Sie Teilaufgabe a) nicht lösen konnten, verwenden Sie Variabeln für allenfalls benötigte Terme in Teilaufgabe b).

Logischer Block	Latenz
PC (PC-Register) lesen (ClkToQ) oder schreiben (Setup Zeit)	40 ps
Instruction memory (Instruktion-Speicher) lesen	400 ps
Add (Addierer)	50 ps
ALU (Arithmetisch-logische Einheit)	200 ps
ALU control (ALU Kontroll-Einheit)	50 ps
Mux (Multiplexer)	5 ps
Registers (Register-Feld) lesen	150 ps
Registers (Register-Feld) schreiben*	100 ps
Data memory (Daten-Speicher) lesen	500 ps
Data memory (Daten-Speicher) schreiben*	0 ps
Sign extend (Vorzeichenerweiterung)	15 ps
Shift left 2 (Logischer Linksshift)	15 ps
AND Gatter	0 ps

Tabelle 4: Latenzen der einzelnen Blöcke des Prozessors in Abb. 1
**Die Schreib-Latenz des Register-Felds und des Daten-Speichers bezieht sich auf die minimal benötigte Zeit zwischen den jeweiligen Eingangssignalen und der Taktflanke.*

(a) (7 Punkte) Nehmen Sie an, die Kontroll-Einheit (Control) verursacht keine Latenzen zur Generierung der Signale. Berechnen Sie die minimal zulässige Taktperiode der gegebenen Architektur.

Lösungsvorschlag: Die minimal zulässige Taktperiode ist durch den kritischen Pfad, d.h. den Signal-Pfad mit der grössten Latenz, gegeben.

Kritischer Pfad ist durch Instruktion `lw` gegeben:
PC + IMem + RMR + ALU + DMem + Mux + RMW = 40 +
400 + 150 + 200 + 500 + 5 + 100 = 1395 ps

□

(b) (4 Punkte) Nehmen Sie nun an, dass die Kontroll-Einheit auch Latenzen zur Generierung der Kontroll-Signale verursacht. Wie lange darf die Kontroll-Einheit maximal zur Generierung des Kontroll-Signales *MemWrite* brauchen, ohne dabei die minimal zulässige Taktperiode zu beeinflussen?

Lösungsvorschlag: Die Kontroll-Einheit kann, sobald die aktuelle Instruktion ausgelesen wurde, mit der Signal-Generierung beginnen, d.h. nach (40 + 400) ps. Da MemWrite nur benötigt wird, um in den Speicher zu schreiben (was instantan passiert, da Data memory Schreib Latenz = 0ps), und keine Abhängigkeiten mit nachfolgenden Logik-Blöcken verursacht, muss bis zur

nächsten Takt-Flanke das Signal entsprechend gesetzt werden (= 1, falls `sw`, sonst =0). Dafür sind 1400 (aus **a**) - 440 = 955 ps zur Verfügung.

□

2.4: Pipelining

(maximal 15 Punkte)

In dieser Aufgabe betrachten wir den in der Vorlesung behandelten MIPS Prozessor mit 5-Stufiger Pipeline (IF → ID → EX → MEM → WB). Die Architektur besitzt folgende Eigenschaften:

- Unterstütztes Forwarding: EX → EX und MEM → EX.
- Alle Sprungentscheidungen werden perfekt vorausgesagt, d.h. nach einer Branch-Instruktion wird immer die korrekte Instruktion in die Pipeline geladen.
- Es wird kein Branch-Delay-Slot verwendet.
- Taktperiode: 100 ns

Gegeben ist folgende C-Funktion `foo`, wobei `unsigned int n` die Länge des Arrays `arr` ist und $n > 0$ gilt:

```
1 int foo(int *arr, unsigned int n){
2     int i = 0;
3     for (i=0;i<n;i++){
4         arr[i] += i*arr[i];
5     }
6 }
```

Für den obigen Prozessor ist die folgende Umsetzung in Assembler gegeben. Dabei gilt beim Funktionsaufruf Register `$a0` = erste Adresse des Arrays `int* arr` und Register `$a1` = `unsigned int n`:

```
1 foo:    move   $t0, $zero
2 loop:   addi   $t0, $t0, 1
3         lw     $t1, 0($a0)
4         mul    $t1, $t1, $t0
5         sw     $t1, 0($a0)
6         addi   $a0, $a0, 4
7         bne    $t0, $a1, loop
8         jr     $ra
```

Hinweis: Auf der letzten Seite des Prüfungsbogens finden Sie eine Übersicht von Assemblerbefehlen.

(a) (4 Punkte) In Tabelle 5 ist die zeitliche Belegung der Pipeline für eine Schleifenausführung gegeben. Zeichnen Sie alle Situationen ein, in denen Forwarding verwendet wird. Verwenden Sie dazu Pfeile, d.h. ↓: Pipeline Phase, die ein Argument forwarded, ↑: Pipeline-Phase, die ein Argument via Forwarding erhält.

	1	2	3	4	5	6	7	8	9	10	11	12
move $t0, $zero	IF	ID	EX	MEM	WB							
addi $t0, $t0, 1		IF	ID	EX	MEM	WB						
lw $t1, 0($a0)			IF	ID	EX	MEM	WB					
mul $t1, $t1, $t0				IF	ID	-	EX	MEM	WB			
sw $t1, 0($a0)					IF	-	ID	EX	MEM	WB		
addi $a0, $a0, 4							IF	ID	EX	MEM	WB	
bne $t0, $a1, loop								IF	ID	EX	MEM	WB

Tabelle 5: Zeitliche Belegung der Pipeline für eine Schleifenausführung

Lösungsvorschlag:

	1	2	3	4	5	6	7	8	9	10	11	12
move $t0, $zero	IF	ID	EX ↓	MEM	WB							
addi $t0, $t0, 1		IF	ID	↑ EX	MEM	WB						
lw $t1, 0($a0)			IF	ID	EX	MEM ↓	WB					
mul $t1, $t1, $t0				IF	ID	-	↑ EX ↓	MEM	WB			
sw $t1, 0($a0)					IF	-	ID	↑ EX	MEM	WB		
addi $a0, $a0, 4							IF	ID	EX	MEM	WB	
bne $t0, $a1, loop								IF	ID	EX	MEM	WB

Tabelle 6: Zeitliche Belegung der Pipeline für eine Schleifenausführung: Musterlösung mit Forwarding

□

(b) (1 Punkt) Im 6. Zyklus muss die Pipeline aufgrund eines Hazards gestallt werden. Um welchen Hazard handelt es sich dabei?

Lösungsvorschlag: Daten-Hazard

□

(c) (2 Punkte) Nehmen Sie nun an $n = 4$. Wie lange braucht die Architektur um das gegebene Assembler Programm auszuführen?

Lösungsvorschlag: Aus Tabelle 5 ist ersichtlich, dass der erste Schleifendurchlauf 12 Zyklen benötigt. Es folgen 3 weitere Durchläufe, die jeweils 6 + 1 (wegen Stall) Zyklen benötigen. Zum Schluss wird noch 1 Zyklus für die jr-Instruktion verwendet.

Somit beträgt die Latenz = $(12+3\cdot7+1)\cdot100\,ns = 3.4\,\mu s$

□

(d) (8 Punkte) Nehmen Sie nun an, dass n immer ein ganzzahliges Vielfaches von 2 ist. Optimieren Sie die Ausführungszeit des Programms, indem Sie loop-unrolling mit einem Entfaltungsfaktor 2 anwenden. Es steht Ihnen dabei ein zusätzliches Register `$t2` zur Verfügung. Berechnen Sie wiederum die Ausführungszeit des durch loop-unrolling optimierten Programms in Abhängigkeit zu n.

Lösungsvorschlag:

```
1  loop:
2          lw $t1, 0($a0)
3          lw $t2, 4($a0)
4          addi $t0, $t0, 1
5          mult $t1, $t1,$t0
6          addi $t0, $t0, 1
7          mult $t2,$t2,$t0
8          sw $t1, 0($a0)
9          sw $t2, 4($a0)
10         addi $a0,$a0,8
11         bne $t0, $a1, loop
```

Ausführungszeit: Die loop wird n/2 mal ausgeführt, wobei jeweils 10 Instruktionen ausgeführt werden pro loop-Durchlauf. Dazu kommen noch je 1 Instruktion zu Beginn und Schluss: move und jr. Somit werden

$$\frac{n}{2}\cdot 10+2$$

Instruktionen ausgeführt, die in der Pipeline

$$\left((5-1)+\frac{n}{2}\cdot 10+2)\right)\cdot 100\,\text{ns}$$

Ausführungszeit benötigen. Z.B. für n = 4, Ausführungszeit = $2.6\mu s$ (nur als Beispiel, nicht konkret gefragt!)

□

Aufgabe 3 : Cache und Virtual Memory (maximal 29 Punkte)

3.1: Grundlagen (maximal 3 Punkte)

Treffen die folgenden Aussagen für die in der Vorlesung behandelten Speicherarchitekturen zu? Kreuzen Sie die richtigen Antworten an. Eine korrekte Antwort gibt 0.5 Punkte. Für eine falsche Antwort werden 0.5 Punkte abgezogen. Die Aufgabe gibt keine negative Gesamtsumme.

- Ein Cache Treffer (hit) erfolgt immer bei übereinstimmenden Tags.
  ☐ richtig ☐ falsch

  Lösungsvorschlag: Falsch, nur wenn Daten auch gültig sind. Valid-Bit muss berücksichtigt werden

  ☐

- Aufeinanderfolgendes Schreiben innerhalb eines Cache-Blockes führt bei einem auf Zurückkopieren ("write back") basierendem Cache dazu, dass dieser Cache-Block mehrmals hintereinander in den Hauptspeicher kopiert wird.
  ☐ richtig ☐ falsch

  Lösungsvorschlag: Falsch, der Block wird erst in den Hauptspeicher zurückgeschrieben wenn dieser Block im Cache ersetzt wird

  ☐

- Snooping Protokolle werden implementiert um Cache-Kohärenz zu ermöglichen.
  ☐ richtig ☐ falsch

  Lösungsvorschlag: Richtig

  ☐

- Der Adressbereich des virtuellen Speichers kann grösser sein als der Adressbereich des verfügbaren Hauptspeichers.
  ☐ richtig ☐ falsch

  Lösungsvorschlag: Richtig

  ☐

- Der Seitenoffset eines Datums wird bei der Übersetzung von virtuellen Adressen in physikalische Adressen aus der Seitentabelle gelesen.
  ☐ richtig ☐ falsch

  Lösungsvorschlag: Falsch, der Seitenoffset ist bei der virtuellen und physikalischen Adresse identisch

  ☐

- Ein TLB Miss tritt ein wenn sich die Seite nicht im Hauptspeicher befindet oder wenn sich die Seite im Hauptspeicher befindet, jedoch der Eintrag im TLB fehlt.
  ☐ richtig ☐ falsch

  Lösungsvorschlag: Richtig

  ☐

3.2: Cache und Busbreite

(maximal 12 Punkte)

Gegeben ist ein Computer-System mit den folgenden Eigenschaften

1. Für Instruktionen und Daten werden getrennte Caches genutzt.
2. Die Speicher- und Busbreite beträgt 1 Wort.
3. Jeder Cache Block ist zwei Wörter lang, der gesamte Block wird bei einem Cache Miss aus dem Hauptspeicher gelesen. Bei einem Write-Miss wird der Block zuerst in den Daten-Cache geschrieben und dann durch die Write-Operation überschrieben.
4. Der Daten-Cache hat eine Treffer (hit) Rate von 95%.
5. 75% aller Speicherzugriffe auf den Daten-Cache sind Lese-Zugriffe, 25% der Speicherzugriffe auf den Daten-Cache sind Schreib-Zugriffe.
6. Im Daten-Cache sind stets 30% aller Blöcke modifiziert (Dirty-Bit ist gesetzt).

Hinweis: Alle Aufgaben können unabhängig von einander gelöst werden.

(a) (6 Punkte) Berechnen Sie die durchschnittliche Anzahl Hauptspeicherzugriffe pro Daten-Cache Zugriff für einen auf Zurückkopieren ("write back") basierenden Cache.

Lösungsvorschlag:

Zugriffs Fall	Speicherzugriffe	Wahrscheinlichkeit
Cache Hit	0 read, 0 write	95%
Cache Miss, ersetzen von unverändertem Block	2 read	5% · 70%
Cache Miss, ersetzen von verändertem Block	2 read, 2 write	5% · 30%

$\Rightarrow 5\% \cdot 70\% \cdot 2 + 5\% \cdot 30\% \cdot (2+2) = 0.07 + 0.06 = 0.13$, im Durchschnitt 0.13 Hauptspeicherzugriffe pro Daten-Cache Zugriff.

Zugriffs Fall	Speicherzugriffe	Wahrscheinlichkeit
Read Hit	0 read, 0 write	75% · 95%
Read Miss, ersetzen von verändertem Block	2 read, 2 write	75% · 5% · 30%
Read Miss, ersetzen von unverändertem Block	2 read	75% · 5% · 70%
Write Hit	0 read, 0 write	25% · 95%
Write Miss, ersetzen von verändertem Block	2 read, 2 write	25% · 5% · 30%
Write Miss, ersetzen von unverändertem Block	2 read	25% · 5% · 70%

$\Rightarrow 75\% \cdot 5\% \cdot 30\% \cdot (2+2) + 75\% \cdot 5\% \cdot 70\% \cdot 2 + 25\% \cdot 5\% \cdot 30\% \cdot (2+2) + 25\% \cdot 5\% \cdot 70\% \cdot 2 = 0.045 + 0.0525 + 0.015 + 0.0175 = 0.13$

□

(b) (4 Punkte) Berechnen Sie die durchschnittliche Anzahl Hauptspeicherzugriffe pro Daten-Cache Zugriff für einen auf durchgängigen Schreiben ("write through") basierenden Cache. Nehmen Sie an, dass bei einem Write nur das veränderte Wort im Hauptspeicher aktualisiert wird und nicht der gesamte Block. Beachten Sie weiterhin Punkt 3 der Beschreibung des Computer-Systems.

Lösungsvorschlag:

Zugriffs Fall	Speicherzugriffe	Wahrscheinlichkeit
Read Hit	0 read, 0 write	95% · 75%
Read Miss	2 read	5% · 75%
Write Hit	0 read, 1 write	95% · 25%
Write Miss	2 read, 1 write	5% · 25%

$\Rightarrow 5\% \cdot 75\% \cdot 2 + 95\% \cdot 25\% \cdot 1 + 5\% \cdot 25\% \cdot (2+1) = 0.075 + 0.2375 + 0.0375 = 0.35$, im Durchschnitt 0.35 Hauptspeicherzugriffe pro Daten-Cache Zugriff.

□

Nehmen Sie nun an, dass das System zusätzlich folgende Eigenschaften aufweist

- Für einen perfekten Instruktions-Cache und perfekten Daten-Cache beträgt die mittlere Zahl der Taktzyklen pro Instruktion (CPI) 2 Zyklen. Der Instruktions-Cache wird weiterhin als perfekt an-genommen.
- 30% der Instruktionen greifen auf den Daten-Cache zu.
- Ein Bustransfer besteht aus einem Zyklus zur Übertragung der Adresse, 10 Zyklen für jeden Hauptspeicherzugriff und einem Zyklus pro Datentransfer.

(c) (2 Punkte) Nehmen Sie nun an, dass die mittlere Anzahl Hauptspeicherzugriffe pro Daten-Cache Zugriff 0.2 ist. Berechnen Sie die mittlere Anzahl Taktzyklen pro Instruktion (CPI).

Lösungsvorschlag: $CPI_{average} = CPI_{perfekt} +$ % Speicherzugriffe · Hauptspeicherzugriffe pro Instruktion · Miss Strafe
Miss Strafe = 12 Zyklen pro Wort
$\Rightarrow CPI_{average} = 2 + 0.3 \cdot 0.2$ Wörter/Instr $\cdot 12$ Zyklen/Wort $= 2.72$ Zyklen

□

3.3: Virtual memory und TLB (maximal 14 Punkte)

Gegeben ist ein Computer-System das einen virtuellen Speicher und einen Translation Lookaside Buffer (TLB) verwendet. Das System besitzt keinen Cache.

Die virtuelle Adresse hat eine Länge von 36 Bits und die physikalische Adresse eine Länge von 31 Bits. Die physikalische und virtuelle Seitengrösse beträgt jeweils 4 KByte. Der TLB ist 16-fach assoziativ und hat 64 Einträge.

Nehmen Sie zudem an, dass der virtuelle und physikalische Speicher Byte-adressiert sind.

Hinweis: 1 GByte = 1024 MBytes, 1 MByte = 1024 KBytes, 1 KByte = 1024 Bytes

(a) (1 Punkt) Berechnen Sie die Anzahl Bits, welche für den Page Offset verwendet werden.

Lösungsvorschlag: 4 KByte Seite = 2^{12} Bytes $\Rightarrow$ 12 Bits werden verwendet für den Page Offset

☐

(b) (1 Punkt) Berechnen Sie die Anzahl Einträge in der Seitentabelle.

Lösungsvorschlag: virtuelle Addresse 36 Bit lang, 12 Bits sind für den Page Offset $\Rightarrow$ 24 Bits für die Seitenaddressierung. 2^{24} Seiten $\Rightarrow$ 2^{24} Einträge in der Seitentabelle.

☐

(c) (1 Punkt) Neben der physikalischen Seitennummer beinhaltet die Seitentabelle pro Eintrag auch noch ein Valid-Bit und ein Dirty-Bit. Andere Statusbits (LRU, Zugriffsberechtigung, ect.) sind in dieser Teilaufgabe nicht zu berücksichtigen. Berechnen Sie die gesamte Grösse der Seitentabelle.

Lösungsvorschlag: physikalische Seitennummer: 19 Bits (31 Bits minus Anzahl Bits für Page Offset) 2^{24} Einträge · (19 (phys. Seitennummer) + 1 (Dirty) + 1 (Valid)) = $21 \cdot 2^{24}$ Bits = $\frac{21}{8} \cdot 2^{24}$ Bytes

☐

(d) (4 Punkte) Wie gross ist ein Eintrag in dem TLB (in Anzahl Bits)? Nehmen Sie dazu an, dass ein Valid-Bit und Dirty-Bit Teil des Eintrags sind. Andere Statusbits (LRU, Zugriffsberechtigung, ect.) sind in dieser Teilaufgabe nicht zu berücksichtigen.

Lösungsvorschlag: 16-fach assoziativer TLB hat 4 Sets bei 64 Einträgen ($\frac{64}{16} = 4$) ⇒ 2 Bits sind notwendig für den TLB index ($4 = 2^2$)
Verbleibende Bits der virtuellen Addresse sind Tag: 36 - 2(index)- 12(Page Offset)=22 Bits
Physikalische Seitennummer ist 19 Bits lang
⇒ Ein Eintrag ist 22 Bit (Tag) + 19 Bit (phys. Seitennummer) + 1 (Dirty) +1 (Valid) = 43 Bits

□

(e) (3 Punkte) Wie viele Stufen sind in einer hierarchischen Seitentabelle notwendig, unter der Voraussetzung, dass alle Seitentabellen in Seiten (Grösse von 4 KByte) passen müssen? Nehmen Sie einfachheitshalber an, dass jeder Tabelleneintrag auf allen hierarchischen Stufen jeweils 4 Bytes beansprucht.

Lösungsvorschlag: Der gesamte virtuelle Speicher ist 2^{36} Bytes gross. Eine Seite hat 2^{12} Bytes. Es passen 2^{10} Seiteneinträge in eine Seite ($\frac{2^{12}}{4}$), diese Seiteneinträge addressieren $2^{10} \cdot 2^{12}$ (Seitengrösse) = 2^{22}Bytes des virtuellen Speichers. Zweite Stufe notwendig, zweite Stufe fügt weitere 2^{10} Seiteneinträge von Seitentabellen hinzu. Mit zwei Stufen können $2^{10} \cdot 2^{10} \cdot 2^{12} = 2^{32}$ Bytes des virtuellen Speichers addressiert werden. Noch eine dritte Stufe notwendig, damit könnten 2^{42} Bytes addressiert werden. Drei Stufen sind notwendig.
Die Seitentabelle braucht insgesamt 2^{24} Einträge. 2^{10} Einträge haben in einer Seite Platz. Auch möglich über Anzahl Einträge anstatt durch adressierbaren Speicherbereich auszurechnen.

□

(f) (1 Punkt) Wie viele Hauptspeicherzugriffe sind bei einem TLB Miss notwendig um auf ein Datum zuzugreifen? Nehmen Sie dazu an, dass es sich um eine zweistufige Seitentabelle handelt und dass die Seitentabellen und Seitentabelleneinträge welche referenziert werden, existieren und sich im Hauptspeicher befinden.

Hinweis: Diese Aufgabe ist unabhängig von Teilaufgabe (e).

Lösungsvorschlag: 3 Zugriffe: zwei Zugriffe für die zwei Seitentabellen ("page table walk") und ein Zugriff auf das Datum.

□

Um zu bestimmen welche Seite bei einem Seitenfehler ersetzt wird, ist die Approximation des Least Recently Used (LRU) Algorithmus mittels Referenz-Bit (second chance) implementiert. Zu jeder Seite gehört ein Referenz-Bit, welches vereinfacht festhält, wie lange diese Seite nicht mehr referenziert wurde.

Hinweis: Der Zeiger ändert sich nur im Falle eines Seitenfehlers

(g) (3 Punkte) Nehmen Sie an, dass der Hauptspeicher nur 3 physikalische Seiten beinhalten kann. Zu Beginn befindet sich das System im Anfangszustand (abgebildet in der ersten Tabelle). Es wird nun in der abgebildeten Reihenfolge auf folgende Seiten zugegriffen:

1, 4, 2

Füllen Sie eine Tabelle pro Speicherzugriff aus. Markieren Sie die momentane Position des Zeigers und füllen Sie die Werte aller Referenz-Bits und die sich im Speicher befindenden Seiten aus.

Zeiger	Ref. Bit	Seite
	1	0
	1	4
⇒	1	2

Anfangszustand

Zeiger	Ref. Bit	Seite

Nach dem Zugriff auf 1

Zeiger	Ref. Bit	Seite

Nach dem Zugriff auf 4

Zeiger	Ref. Bit	Seite

Nach dem Zugriff auf 2

Lösungsvorschlag:

□

Zeiger	Ref. Bit	Seite
	1	0
	1	4
⇒	1	2

Anfangszustand

Zeiger	Ref. Bit	Seite
⇒	0	0
	0	4
	1	1

Nach dem Zugriff auf 1

Zeiger	Ref. Bit	Seite
⇒	0	0
	1	4
	1	1

Nach dem Zugriff auf 4

Zeiger	Ref. Bit	Seite
	1	2
⇒	1	4
	1	1

Nach dem Zugriff auf 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`
`beqz $s1, label`	Wenn `($s1 == 0)`, verzweige zu `label`
`bne $s1, $s2, 25`	Wenn `($s1 != $s2)`, verzweige zu `PC + 4 + 100`
`bnez $s1, label`	Wenn `($s1 != 0)`, verzweige zu `label`
`div $s1, $s2`	Teile `$s1` durch `$s2` und speichere den Quotienten in `$LO` und den Restbetrag in `$HI`
`j label`	Springe zu `label`
`jal label`	Springe zu `label` und setze `$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]`
`mfhi $s1`	`$s1 = $HI`
`mflo $s1`	`$s1 = $LO`
`move $s1, $s2`	`$s1 = $s2` (Pseudoinstruktion)
`mul $s1, $s2, $s3`	`$s1 = $s2 * $s3` (Multiplikation)
`nop`	No operation
`or $s1, $s2, $s3`	`$s1 = $s2 | $s3` (bitweises Oder)
`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`
`srl $s1, $s2, 10`	`$s1 = $s2 >> 10`
`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`
`sb $s1, 100($s2)`	`Speicher[$s2 + 100] = $s1` (ein Byte)
`sw $s1, 100($s2)`	`Speicher[$s2 + 100] = $s1` (ein Wort)