Aufgabe 1: Stack – Funktionsweise

Erläutern Sie die Funktionsweise eines Stacks bzw. Kellerspeichers anhand des folgenden Pseudocodes. Tragen Sie die nach Ablauf der Befehlssequenz resultierenden Werte entsprechend ein (vgl. Foliensatz 15 – Befehlssatz, Folie 16ff). Geben Sie alle Registerinhalte in **hexadezimaler** Notation an.

```
01:  SP ← (FFFF)16
02:  R1 ← 3
03:  R2 ← lsh(R1+1)
04:  push(R2)
05:  R1 ← lsh(R2+1)
06:  pop(R3)
07:  R2 ← R2+R3
08:  push(R1)
09:  R1 ← R1-R2
10:  push(R2)
11:  R2 ← lsh(R2+4)
12:  pop(R1)
13:  R3 ← R1+R2
14:  pop(R1)
15:  R2 ← lsh(R1)
16:  push(R3)
17:  R3 ← R3+R2
18:  push(R1)
19:  push(R2)
```

12	R1
24	R2
5C	R3
FFFC	SP

Register

	$(FFFB)_{16}$
	$(FFFC)_{16}$
24	$(FFFD)_{16}$
12	$(FFFE)_{16}$
38	$(FFFF)_{16}$

RAM

Zeit	Register				RAM			
	R1	R2	R3	SP	FFFC	FFFD	FFFE	FFFF
1				FFFF				
2	3			FFFF				
3	3	8		FFFF				
4	3	8		FFFE				8
5	12	8		FFFE				8
6	12	8	8	FFFF				(8)
7	12	10	8	FFFF				(8)
8	12	10	8	FFFE				12
9	2	10	8	FFFE				12
10	2	10	8	FFFD			10	12
11	2	28	8	FFFD			10	12
12	10	28	8	FFFE			(10)	12
13	10	28	38	FFFE			(10)	12
14	12	28	38	FFFF			(10)	(12)
15	12	24	38	FFFF			(10)	(12)
16	12	24	38	FFFE			(10)	38
17	12	24	5C	FFFE			(10)	38
18	12	24	5C	FFFD			12	38
19	12	24	5C	FFFC		24	12	38

Ein Stack ist eine Datenstruktur, welche nach dem Last In First Out Prinzip arbeitet. Elemente werden aufeinandergestapelt. Man denke hierbei an einen zweidimensionalen Rucksack oder ein Kartendeck. Es gibt hier zwei Operationen, **push** und **pop**.

Push legt ein Element auf den Stapel, pop entfernt eines aus diesem.

Das mit push auf den Stapel gelegte Element ist das erste, welches mit pop zurückgeliefert wird.

Aufgabe 2: Stack – Auflösen von Klammerstrukturen

Verwenden Sie zwei Stacks, um den folgenden logischen Ausdruck auszuwerten:

$$((a \to b)_1 \vee ((c \oplus (b \wedge (\neg(a \vee d)_2)_3)_4)_5 \vee (d \oplus c)_6)_7)_8$$

Es gibt einen Stack für Operatoren und einen für Operanden. Nach Initialisierung und vor Beginn der Auswertung zeigt der Stack-Pointer des Operatoren-Stacks auf die Adresse $(FFFF)_{16}$, der des Operanden-Stacks auf $(FFF1)_{16}$. Die Operatoren und Operanden werden symbolisch in der Reihenfolge ihres Einlesens auf den jeweiligen Stack gelegt (also $\vee, \oplus$, etc. auf den Operatoren-Stack $a, b, a \oplus b$, etc. auf den Operanden-Stack).

Gehen Sie bei der Auswertung wie folgt vor:

- Lesen Sie den Ausdruck symbolweise ein. Der Ausdruck ist bereits der Rangfolge der Operatoren entsprechend korrekt geklammert. Öffnende Klammern werden ignoriert.
- Sobald eine schließende Klammer eingelesen wird, ist der oberste Operator vom Stack zu nehmen. Dem Operator entsprechend wird eine bestimmte Anzahl Operanden vom Operanden-Stack genommen und miteinander verknüpft.
- Das Ergebnis wird wieder auf den Operanden-Stack gelegt.

Vervollständigen Sie die Tabelle. Tragen Sie die Inhalte der beiden Stacks ein und geben zu den Einträgen die entsprechende 16-bit Adresse hexadezimal an. Stellen Sie jeweils den Zustand der beiden Stacks unmittelbar *vor* dem Auswerten der angegebenen schließenden Klammer dar.

	Operatoren-Stack		**Operanden-Stack**	
	Adresse	Inhalt	Adresse	Inhalt
	$FFFF$		$FFF1$	
$)_1$			$FFF0$	b
	$FFFF$	$\to$	$FFF1$	a
$)_2$	$FFFB$	$\vee$	$FFED$	d
	$FFFC$	$\neg$	$FFEE$	a
	$FFFD$	$\wedge$	$FFEF$	b
	$FFFE$	$\oplus$	$FFF0$	c
	$FFFF$	$\vee$	$FFF1$	$(a \to b)$
$)_3$	FFFC FFFD FFFE FFFF	¬ ∧ ⊕ ∨	FFEE FFEF FFF0 FFF1	(a ∨ d) b c (a → b)
$)_4$	FFFD FFFE FFFF	∧ ⊕ ∨	FFEE FFEF FFF0 FFF1	¬(a∨d) b c (a→b)
$)_5$	FFFE FFFF	⊕ ∨	FFEF FFF0 FFF1	b∧(¬(a∨d)) c (a→b)
$)_6$	FFFD FFFE FFFF	⊕ ∨ ∨	FFEE FFEF FFF0 FFF1	g c⊕(b∧(¬(a∨d))) (a→b)
$)_7$	FFFE FFFF	∨ ∨	FFEF FFF0 FFF1	(d⊕c) c⊕(b∧(¬(a∨d))) (a→b)
$)_8$	FFFF	∨	FFF0 FFF1	c⊕(b∧¬(a∨d)))∨(d⊕c) (a→b)
	$FFFF$		$FFF1$	$(a \to b) \vee (c \oplus (b \wedge \neg(a \vee d))) \vee (d \oplus c)$

Aufgabe 3: Stack – Funktionsaufrufe

Gegeben ist nachfolgendes Programm in Pseudocode-Notation. Die Ausführung startet bei `Program Start()`, alle Variablen sind global deklariert.

a) Erklären Sie, wie ein Stack bei Funktionsaufrufen Verwendung findet.

festhalten der Rücksprungadresse

b) Tragen Sie in die Tabelle die aktuellen Werte der Variablen und den Inhalt des Stacks *nach* Durchführung jeder Instruktion ein. Tragen Sie weiters in die Tabelle ein, an welchen Stellen der Stack mit welchen Operationen (push/pop) verändert wird. Die Adressen (`0:` bis `14:`) der Instruktionen sind jeweils links neben dem Pseudocode angegeben. Die ersten Zeilen sind bereits vorausgefüllt.

```
   Program Start() {
0:   b = 0;
1:   f = 1;
2:   c = 1;
3:   count();
4:   exit;
   }

   function count() {
5:    if (f % 5 == 0) {
6:       minus();
      } else {
7:       b = f;
8:       f = f + c;
9:       c = b;
      }
10:   if (f > 0) {
11:      count();
      }
12:    return;
   }

   function minus() {
13:   f = 0;
14:   return;
   }
```

Adresse	f	b	c	Stack-Operation	Stack-Inhalt
0		**0**			
1	**1**	**0**			
2	**1**	**0**	**1**		
3	**1**	**0**	**1**	**push(4)**	4
5	1	0	1		4
7	1	1	1		4
8	2	1	1		4
9	2	1	1		4
10	2	1	1		4
11	2	1	1	push(12)	12,4
5	2	1	1		12,4
7	2	2	1		12,4
8	3	2	1		12,4
9	3	2	2		12,4
10	3	2	2		12,4
11	3	2	2	push(12)	12,12,4
5	3	2	2		12,12,4
7	3	3	2		12,12,4
8	5	3	2		12,12,4
9	5	3	3		12,12,4
10	5	3	3		12,12,4
11	5	3	3	push(12)	12,12,12,4
5	5	3	3		12,12,12,4

6	5	3	3	push(10)	10,12,12,4
13	0	3	3		10,12,12,4
14	0	3	3	pop()	12,12,12,4
10	0	3	3		12,12,12,4
12	0	3	3	pop()	12,12,4
12	0	3	3	pop()	12,4
12	0	3	3	pop()	4
4	0	3	3		

Aufgabe 4: Adressierungsverfahren – Speicherzugriffssequenz

Gegeben ist der folgende Ausschnitt aus dem Speicher eines Computers:

	Adresse	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	Startwerte	5	3	B	1	5	C	A	F	F	C	B	1	4	7	E	C
X	Endwerte	5	2	C	E	2	4	C	A	F	C	B	1	4	7	5	C

Auf diesem Computer wird folgende Befehlsfolge ausgeführt (Speicherzugriffsbefehle siehe *Einführung in die Technische Informatik*, Seite 154ff):

1	R0	←	*memory*[4]
2	R5	←	*memory*[2]
3	R4	←	*memory*[1]
4	*memory*[+(R4)]	←	–(R5)
5	R1	←	R5 + 4
6	*memory*[R1]	←	R1
7	*memory*[R0 + 1]	←	*memory*[R0]
8	*memory*[–(R4)]	←	*memory*[*memory*[R1]]
9	*memory*[*memory*[R4]]	←	R0–
10	*memory*[R1]	←	R0
11	*memory*[–R1]	←	*memory*[R0]
12	*memory*[R4 + 2]	←	R5+
13	R2	←	1
14	R4	←	*memory*[R0 + 2]
15	*memory*[R4]	←	+R2
16	*memory*[(R5)+]	←	(R2)+
17	*memory*[R3 + 1]	←	R1 – 1

Tragen Sie den Inhalt der Register nach jedem Befehl in die folgende Tabelle ein. Falls Speicherzugriffe erfolgen, geben Sie in der Spalte *Speicherzugriff(e)* die Art des Speicherzugriffes (rd(Adresse)/wr(Adresse)) und die zugehörige Adresse an. Alle Register wurden mit 0 initialisiert.

Befehl	R0	R1	R2	R3	R4	R5	Speicherzugriff(e)
1	5	0	0	0	0	0	rd(4) → 5
2	5	0	0	0	0	B	rd(2) → B
3	5	0	0	0	3	B	rd(1) → 3
4	5	0	0	0	4	A	wr(4) ← A
5	5	E	0	0	4	A	0
6	5	E	0	0	4	A	wr(E) ← F
7	5	E	0	0	4	A	rd(5) → C, wr(6) ← C
8	5	E	0	0	3	A	rd(E) → E, rd(E) → E, wr(3) ← E
9	4	E	0	0	3	A	rd(9) → E, rd(E) → E, wr(E) ← 5
10	4	E	0	0	3	A	rd(E) → 5, wr(5) ← 4
11	4	D	0	0	3	A	rd(4) → A, rd(D) → 7, wr(3) ← A
12	4	D	0	0	3	B	rd(5) → 4, wr(4) ← A
13	4	D	1	0	3	B	0
14	4	D	1	0	C	B	rd(6) → C
15	4	D	2	0	C	B	rd(C) → 4, wr(4) ← 2
16	4	D	3	0	C	C	rd(B) → 1, wr(1) ← 2
17	4	D	3	0	C	C	rd(1) → 2, wr(2) ← C

Aufgabe 5: Pipeline – Produktion

In einer Fabrik werden verschiedene Produkte in einer Pipeline gefertigt. Dabei müssen verschiedene Tasks (Task1-Task5) abgearbeitet werden. Abhängig vom Produkt, benötigen die einzelnen Tasks mehr oder weniger Zeit – gemessen in generischen Zeiteinheiten (ZE):

Produktart	**Task1 [ZE]**	**Task2 [ZE]**	**Task3 [ZE]**	**Task4 [ZE]**	**Task5 [ZE]**
Produkt1	1	2	1	1	1
Produkt2	1	1	1	2	1
Produkt3	1	1	1	1	1

In der ersten Station (Task1) wird eine Teilbearbeitung eines Produkt durchgeführt. Danach wandert das Produkt weiter zum Task2 und gleichzeitig wird das nächste Produkt in Task1 bearbeitet. Sobald ein Produkt in Task2 fertig bearbeitet wurde, wandert es weiter zu Task3 und das zuvor bearbeitete Produkt wandert von Task1 zu Task2. D.h., sobald die Bearbeitung eines Produkts in einem Task abgeschlossen ist, wandert es weiter zum nächsten Task, sofern das zuvor bearbeitete Produkt bereits weitergewandert ist (es gibt also keinen globalen Takt).

a) Zeichnen Sie den Ablauf in der Pipeline, wenn Produkte in nachfolgender Reihenfolge produziert werden und die Pipeline für die Produktion ein- und danach wieder auslaufen muss.

Produkt3 - Produkt2 - Produkt2 - Produkt1 - Produkt1 - Produkt3

b) Wie lange dauert die Produktion der Produkte in der oben genannten Reihenfolge insgesamt? Wie lange würde die Produktion dauern, wenn kein *Pipelining* angewendet werden würde? Welche Zeitersparnis ergibt sich durch Pipelining?

c) Nehmen Sie an, dass die Produktion der Produkte ohne Pipelining im Durchschnitt 6 Zeiteinheiten benötigt. Um wieviel Prozent steigt die Produktivität (=fertige Produkte/ZE) durch Pipelining wenn Sie genau die Produkte aus Unteraufgabe a) in der gegeben Reihenfolge produzieren? Berücksichtigen Sie auch die Ein- und Auslaufzeit der Pipeline.

d) Welche weitere Produktivitätssteigerung kann erreicht werden, wenn die Produkte in obiger Reihenfolge rund um die Uhr im Schichtbetrieb produziert werden, d.h. wenn das Ein und Auslaufen der Pipeline vernachlässigt werden kann?

Zeit	T1	T2	T3	T4	T5
1	$P3_1$				
2	$P2_1$	$P3_1$			
3	$P2_2$	$P2_1$	$P3_1$		
4	$P1_1$	$P2_2$	$P2_1$	$P3_1$	
5	$P1_2$	$P1_1$ [1/2]	$P2_2$	$P2_1$ [1/2]	$P3_1$
6	($P1_2$)	$P1_1$ [2/2]	($P2_2$)	$P2_1$ [2/2]	
7	$P3_2$	$P1_2$ [1/2]	$P1_1$	$P2_2$ [1/2]	$P2_1$
8	($P3_2$)	$P1_2$ [2/2]	($P1_1$)	$P2_2$ [2/2]	
9		$P3_2$	$P1_2$	$P1_1$	$P2_2$
10			$P3_2$	$P1_2$	$P1_1$
11				$P3_2$	$P1_2$
12					$P3_2$

a) Produkt 1_{1-2} → $P1_{1-2}$

Produkt 2_{1-2} → $P2_{1-2}$

Produkt 3_{1-2} → $P3_{1-2}$

Geklammerter Ausdruck → Stall

b) Mit Pipelining: 12 Zeiteinheiten

Ohne Pipelining: 5 + 6 + 6 + 6 + 6 + 5 = 34 Zeiteinheiten

Differenz: 34 - 12 = 22 Zeiteinheiten

c) Ohne Pipelining: 6 * 6 Zeiteinheiten / 6 Produkte = Durchschnittlich **6** Zeiteinheiten pro Produkt

Mit Pipelining: 12 Zeiteinheiten / 6 Produkte = Durchschnittlich **2** Zeiteinheiten pro Produkt

6 Zeiteinheiten pro Produkt / 2 Zeiteinheiten pro Produkt = 3 → Steigerung um +2 ~~3~~00%

d)

Zeit	T1	T2	T3	T4	T5
1	$P3_1$				
2	$P2_1$	$P3_1$			
3	$P2_2$	$P2_1$	$P3_1$		
4	$P1_1$	$P2_2$	$P2_1$	$P3_1$	
5	$P1_2$	$P1_1$ [1/2]	$P2_2$	$P2_1$ [1/2]	$P3_1$
6	($P1_2$)	$P1_1$ [2/2]	($P2_2$)	$P2_1$ [2/2]	
7	$P3_2$	$P1_2$ [1/2]	$P1_1$	$P2_2$ [1/2]	$P2_1$
8	($P3_2$)	$P1_2$ [2/2]	($P1_1$)	$P2_2$ [2/2]	
9	**$P3_3$**	$P3_2$	$P1_2$	$P1_1$	$P2_2$
10	**$P2_3$**	**$P3_3$**	$P3_2$	$P1_2$	$P1_1$
11	**$P2_4$**	**$P2_3$**	**$P3_3$**	$P3_2$	$P1_2$
12	**$P1_3$**	**$P2_4$**	**$P2_3$**	**$P3_3$**	$P3_2$

➔ 8 Zeiteinheiten pro Set → 8 Zeiteinheiten / 6 Produkte = Durchschnittlich 1.3 Zeiteinheiten pro Produkt

~~34 Zeiteinheiten (ohne Pipelining) / 8 Zeiteinheiten (mit Pipelining) = 4.25 → +325%~~

Steigerung +50% (12|8)

Aufgabe 6: Pipelining – Performanceverbesserung

Ein Prozessor besitzt eine fünfstufige Pipeline: *Fetch*, *Decode*, *Execute*, *Memory* und *Write Back*.

Der Instruktionssatz des Prozessors umfasst die drei Instruktionstypen *i1*, *i2*, *i3* und *i4*. Die Dauer der Ausführung einer Verarbeitungsstufe, abhängig vom Typ der Instruktion, ist in folgender Tabelle angegeben:

Instruktionstyp	Fetch	Decode	Execute	Memory	Write Back	Summe
i1	25ns	50ns	150ns	100ns	0ns	325ns
i2	50ns	50ns	250ns	100ns	50ns	500ns
i3	50ns	75ns	150ns	50ns	125ns	450ns
i4	75ns	125ns	150ns	50ns	0ns	400ns

a) Geben Sie die kleinstmögliche Taktzykluszeit für diesen Prozessor an, wenn die Instruktionen ohne Pipelining ausgeführt werden. Pro Taktzyklus soll genau eine Instruktion ausgeführt werden.

~~325ns → i1~~ Längste! 500ns → i2

b) Der in Unteraufgabe a) verwendete Prozessor soll auf Pipelineverarbeitung umgestellt werden. Aus Kostengründen sollen die Verarbeitungsstufen unverändert bleiben. Wie groß wählen Sie unter dieser Voraussetzung die Taktzykluszeit der Pipeline?

250ns → Execute in i2

c) Berechnen Sie den theoretischen Durchsatz in MIPS für die Prozessoren aus Unteraufgabe a) und b).

Ohne Pipelining: $\frac{1}{\cancel{325}\,500 \cdot 10^{-9}} \cdot 10^{-6} = \frac{1}{\cancel{3,25} \cdot 10^{-7}} \cdot 10^{-6} = \frac{10}{\cancel{3,25}\,5} \approx \cancel{3,08}\ 2$ MIPS

Mit Pipelining: $\frac{1}{250 \cdot 10^{-9}} \cdot 10^{-6} = \frac{1}{2,5 \cdot 10^{-7}} \cdot 10^{-6} = \frac{10}{2,5} = 4$ MIPS

d) Angenommen, bei Pipelining (vgl. Aufgabe b) liegt der reale Durchsatz des Prozessors 25% unter dem theoretischen Durchsatz. Wie viele Instruktionen verlassen in 250ms durchschnittlich die Pipeline?

250ms → 0,25s

$(1-0,25) \cdot 0,25\,s \cdot 4$ MIPS $= 0,75$ MIPS

e) Welche der folgenden Änderungen der Pipelinestruktur bringt den größten Nutzen hinsichtlich des theoretischen Durchsatzes?

(a) Zusammenfassen und Optimieren von *Fetch* und *Decode*, sodass alle Instruktionen in der neuen Stufe *Fetch & Decode* 125ns benötigen.

(b) Auftrennen der Execute-Stufe in zwei Stufen *Execute1* und *Execute2*, wobei i1, i3 und i4 jeweils 75ns in den beiden neuen Stufen benötigen, i2 jeweils 125ns. (eigentlich nicht möglich)

(c) Eine allgemeine Optimierung, die jede Stufe, die mehr als 20ns benötigt, um 20ns verkürzt.

a) Würde lediglich nur die Taktzykluszeit in i4 auf 325ns reduzieren, ~~ändert jedoch nichts am Durchsatz~~. (3,08 MIPS)

b) Eine Reduktion auf 125ns möglich (8 MIPS).

c) Eine Reduktion auf 230ns möglich (4.3 MIPS).

✓

Aufgabe 7: Pipelining – RAW-Hazard

Sie arbeiten mit einem Prozessor, der eine vierstufige Pipeline besitzt: Fetch (F), Decode (D), Execute (E) und Store (S).

Bedingt durch die Pipelinestruktur kann es zu *RAW Data Hazards* kommen, welche durch verzögerte Ausführung (*stall*) der lesenden Instruktion vermieden werden. Dabei wird die lesende Instruktion erst dann in Stufe D verarbeitet, wenn die schreibende Instruktion Stufe S abgeschlossen hat. Nehmen Sie zwecks Vereinfachung an, dass Lesezugriffe auf den Stack ebenfalls in Stufe D und Schreibzugriffe in Stufe S ausgeführt werden. Auf dem Prozessor wird folgendes Programm ausgeführt:

```
ADD   R1, R2, R1    # R1 und R2 addieren, Resultat in R1
PUSH  R2            # R2 auf Stack ablegen
INC   R1            # R1 incrementieren
DIV   R3, R4, R5    # R3 durch R4 dividieren, Resultat in R5
SUB   R5, R1, R6    # R1 von R5 subtrahieren, Resultat in R6
MULT  R5, R6, R1    # R5 mit R6 multiplizieren, Resultat in R2
POP   R3            # oberstes Element des Stacks in R3 ablegen
```

a) Zeichnen Sie die Belegung der Pipeline für das gegebene Programm unter der Voraussetzung, dass die Pipeline am Beginn und am Ende leer ist.

Zeit ↓	F	D	E	S
1	ADD			
2	PUSH	ADD		
3	INC	(PUSH)	ADD	
4	DIV	INC	PUSH	ADD
5	SUB	DIV	INC	PUSH
6	(MULT)	(SUB)	DIV	INC
7	(MULT)	(SUB)		DIV
8	MULT	SUB		
9	(POP)	(MULT)	SUB	
10	(POP)	(MULT)		SUB
11	POP	MULT		
12		POP	MULT	
13			POP	MULT
14				POP

b) Kreuzen Sie nachfolgend an, ob es sich um korrekte Umordnungen der Instruktionsfolge handelt oder nicht. Eine Umordnung ist korrekt, wenn die Funktionalität erhalten bleibt. Begründen Sie Ihre Antwort und geben Sie bei korrekten Umordnungen an, *wie viele Takte* die Ausführung benötigt.

ADD	PUSH	ADD	PUSH
PUSH	ADD	INC	POP
INC	DIV	POP	ADD
POP	INC	DIV	INC
MULT	SUB	SUB	DIV
SUB	MULT	PUSH	SUB
DIV	POP	MULT	MULT
❍ korrekt	☒ korrekt	❍ korrekt	❍ korrekt
☒ nicht korrekt	❍ nicht korrekt	☒ nicht korrekt	☒ nicht korrekt
POP vor DIV	13 Takte	POP vor PUSH	POP vor DIV ✓

Aufgabe 8: Pipelining – Control-Hazard & Branch Prediction

Vergleichen Sie den Ablauf eines Programms auf zwei unterschiedlichen Prozessoren:

- Prozessor A mit 3-stufiger Pipeline: *Fetch (F), Decode & Execute (D/E)* und *Memory & Store (M/S)*.
- Prozessor B mit ~~3~~ 4-stufiger Pipeline: *Fetch (F), Decode (D), Execute (E)* und *Memory & Store (M/S)*.

Auf beiden Prozessoren läuft jeweils das unten links angeführte Programm P. Bei *i1 [if Z goto L2]* wird der Sprung zur Instruktion mit der Marke *L2* erst beim dritten Mal ausgeführt. Der unbedingte Sprung *i2 [goto L1]* wird in Stufe D erkannt und im darauffolgenden Zyklus der nächste Befehl von der Zieladresse geladen.

Beide Prozessoren besitzen eine dynamische Sprungvorhersage: Wird eine Sprungbedingung erstmals erreicht, wird der Sprung als nicht auszuführen angenommen. Im Wiederholungsfall wird angenommen, dass die Auswertung der Sprungbedingung dasselbe Ergebnis wie zuletzt liefert.

Beide Prozessoren benutzen zudem eine sogenannte *Pipeline-Freeze Strategie*: Sobald Stufe D einen bedingten Sprungbefehl erkannt hat, wird die Verarbeitung aller nachfolgenden Befehle eingefroren, bis der Sprungbefehl im Schritt k die Stufe S verlassen hat. Falls die Sprungvorhersage korrekt war, übernimmt die Pipeline im Schritt k den Befehl von Stufe F in Stufe D. Anderenfalls wird im Schritt k der Befehl in Stufe F gelöscht (*Pipeline-Flush*) und stattdessen der als nächstes auszuführende Befehl in Stufe F bearbeitet.

a) Zeichnen Sie den Ablauf der Pipelineverarbeitung (vgl. Foliensatz 16 – Pipelining, Folie 27ff.) für beide Prozessoren. Setzen Sie die Darstellung solange fort, bis alle Instruktionen vollständig abgearbeitet wurden. Die ersten Takte sind bereits vorausgefüllt.

Hinweis: Möglicherweise werden nicht alle Zeilen benötigt.

Programm P:

```
L1:  i0
     i1 [if Z goto L2]
     i2 [goto L1]
L2:  i3
```

Zeit ↓	F	D/E	M/S
1	***i0***		
2	***i1***	***i0***	
3	***i2***	***i1***	***i0***
4	i2		i1
5		i2	
6	i0		i2
7	i1	i0	
8	i2	i1	i0
9	i2		
10		i2	
11	i0		i2
12	i1	i0	
13	i2	i1	i0
14	i2		i1
15	i3		
16		i3	
17			i3
18			
19			
20			

Zeit ↓	F	D	E	M/S
1	***i0***			
2	***i1***	***i0***		
3	***i2***	***i1***	***i0***	
4	i2		i1	i0
5	i2			i1
6		i2		
7	i0		i2	
8	i1	i0		i2
9	i2	i1	i0	
10	i2		i1	i0
11	i2			i1
12		i2		
13	i0		i2	
14	i1	i0		i2
15	i2	i1	i0	
16	i2		i1	i0
17	i2			i1
18	i3			
19		i3		
20			i3	
21				i3
22				

b) Vergleichen Sie die benötigte Taktanzahl der beiden Prozessoren. Was fällt Ihnen in Bezug auf die Anzahl der verlorenen Takte auf?

Bei Prozessor B (mehr Pipeline-Stages) längere Pipeline-Freezes.