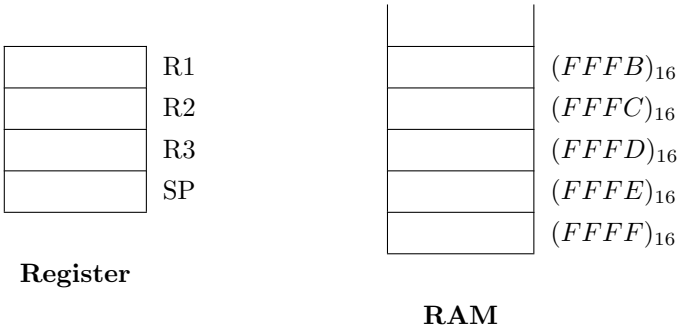
Aufgabe 1: Stack – Funktionsweise

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.

```
SP ← (FFFF)16
R1 ← 1
R2 ← lsh(R1+1)
push(R2)
R1 ← lsh(R2+3)
pop(R3)
R2 ← R2+R3
R3 ← R1-R2
push(R1)
push(R2)
R2 ← lsh(R2+2)
pop(R1)
R3 ← R1+R2
pop(R1)
R2 ← lsh(R1)
R3 ← R3-R2
push(R1)
push(R2)
push(R3)
```



Aufgabe 2: Stack – Auflösen von Klammerstrukturen

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

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

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 $\vee, \oplus, \ldots$ auf den Operatoren-Stack $a, b, a \oplus b, \ldots$ 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
	$FFFF$		$FFF1$	
$)_1$	$FFFF$	$\vee$	$FFF0$	b
			$FFF1$	a
$)_2$	$FFFD$		$FFEF$	
	$FFFE$		$FFF0$	
	$FFFF$	$\wedge$	$FFF1$	$a \vee b$
$)_3$				
$)_4$				
$)_5$				
$)_6$				
$)_7$				
$)_8$				
	$FFFF$		$FFF1$	$((a \vee b) \wedge (\neg(b \oplus c))) \vee ((\neg(b \vee d)) \wedge a)$

Aufgabe 3: Stack – Funktionsaufrufe

Gegeben ist folgendes 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.

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

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

   function count() {
4:    while (b < 5) {
5:       a = a + 1;
6:       if (a % 2 == 0) then {
7:         piff();
         } else {
8:         paff();
         }
      }
9:    return;
   }

   function piff() {
10:   b = a * a - 1;
11:   paff();
12:   return;
   }

   function paff() {
13:   b = b + 1;
14:   return;
   }
```

Adresse	a	b	Stack-Operation	Stack-Inhalt
0		**0**		
1	**1**	**0**		
2	**1**	**0**	**push(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
Wert	8	4	E	7	B	C	5	5	E	E	A	C	7	6	9	C

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

1	R0	←	$memory$[5]
2	R5	←	$memory$[4]
3	R4	←	$memory$[1]
4	$memory$[+(R4)]	←	–(R5)
5	R1	←	R0 + 2
6	$memory$[R1]	←	R1
7	$memory$[R0 + 1]	←	$memory$[R0]
8	$memory$[–(R4)]	←	$memory$[$memory$[R0]]
9	$memory$[$memory$[R4]]	←	R0–
10	$memory$[R0]	←	R0
11	$memory$[–R0]	←	$memory$[R0]
12	$memory$[R4 + 2]	←	R5+
13	R2	←	1
14	R4	←	$memory$[R0 + 2]
15	$memory$[R4]	←	+R1
16	$memory$[(R5)+]	←	(R2)+
17	$memory$[R2 + 1]	←	R2 – 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	**C**	**0**	**0**	**0**	**0**	**0**	rd(5)
2							
3							
4							
5							
6							
7							
8							
9							
10							
11							
12							
13							
14							
15							
16							
17							

Aufgabe 5: Pipeline – Kekse

In einer Bäckerei werden zu Jahresbeginn Kekse wie am Fließband produziert. Dabei muss zuerst der Teig aus den Zutaten zusammen gemischt (M) und ausgewalkt (W) werden. Danach werden die Kekse ausgestochen (A), gebacken (B) und verziert (V). Abhängig von der Keksart, benötigen die einzelnen Stationen mehr oder weniger Zeit, gemessen in generischen Zeiteinheiten (ZE):

Keksart	**Mischen [ZE]**	**Walken [ZE]**	**Ausstechen [ZE]**	**Backen [ZE]**	**Verzieren [ZE]**
Mürbteigkekse	1	1	1	1	1
Linzer Augen	1	1	2	1	1
Lebkuchen	1	1	1	1	2

In der ersten Station wird eine gewisse Masse Teig zusammen gemischt. Danach wandert die Teigmasse weiter zum Walken und gleichzeitig wird die nächste Masse Teig gemischt. Sobald ein Teig fertig gewalkt ist, wandert er weiter in die Station Backen und der zuvor gemischte Teig wird gewalkt. D.h. sobald die Bearbeitung einer Teigmasse in einer Station abgeschlossen ist wandert die Teigmasse zur nächsten Station sobald diese frei geworden ist.

a) Zeichnen Sie den Ablauf am Fließband (=Pipeline), wenn Kekse in nachfolgender Reihenfolge produziert werden und das Fließband für die Produktion hochgefahren und danach wieder heruntergefahren werden muss (d.h. berücksichtigen Sie auch das Ein- und Auslaufen der Pipeline).

Mürbteigkekse - Lebkuchen - Lebkuchen - Linzer Augen - Linzer Augen - Mürbteigkekse

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

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

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

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* und *i3*. 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	50ns	50ns	200ns	50ns	50ns	400ns
i2	50ns	100ns	100ns	100ns	50ns	400ns
i3	50ns	50ns	100ns	50ns	0ns	250ns

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.

b) Der in Teilbeispiel 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?

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

d) Angenommen, bei Pipelining (vgl. Aufgabe b) liegt der reale Durchsatz des Prozessors 40% unter dem theoretischen Durchsatz. Wie viele Instruktionen verlassen in 500ms durchschnittlich die Pipeline?

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* 100ns benötigen.

   (b) Auftrennen der Execute-Stufe in zwei Stufen *Execute1* und *Execute2*, wobei i2 und i3 50ns in jeder der beiden neuen Stufen benötigen, i1 je 150ns.

   (c) Eine allgemeine Optimierung, die jede Stufe, die mehr als 0ns benötigt, um 20ns verkürzt.

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:

```
MULT  R4, R4, R4   # R4 quadrieren, Resultat in R4
SUB   R2, R3, R4   # R4 von R3 subtrahieren, Resultat in R2
PUSH  R4           # R4 auf Stack ablegen
SLL   R6, R2, 1    # Shift left von R2 um eine Stelle, Resultat in R6
ADD   R4, R2, R7   # R2 und R7 addieren, Resultat in R4
POP   R9           # oberstes Element vom Stack in R9 laden
```

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				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				

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.

MULT	MULT	MULT	MULT
SLL	SUB	PUSH	POP
PUSH	ADD	SUB	SUB
SUB	PUSH	POP	PUSH
ADD	POP	SLL	SLL
POP	SLL	ADD	ADD
❍ korrekt	❍ korrekt	❍ korrekt	❍ korrekt
❍ nicht korrekt	❍ nicht korrekt	❍ nicht korrekt	❍ nicht korrekt

Aufgabe 8: Pipelining – Control-Hazard & Branch Prediction

Vergleichen Sie den Ablauf desselben 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 5-stufiger Pipeline: *Fetch (F), Decode (D), Execute (E), Memory (M)* und *Store (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 den 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 12ff.) 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
```

Bei i3 (z.B. Slot 5) eventuell auf branch delay hinweisen.

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

Zeit ↓	F	D	E	M	S
1	***i0***				
2	***i1***	***i0***			
3	***i2***	***i1***	***i0***		
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					
21					
22					
23					
24					
25					
26					

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