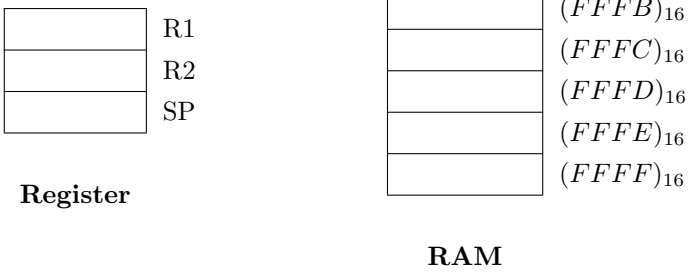
Aufgabe 1: Stack – Funktionsweise

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 11 – Befehlssatz, Folie 16ff). Bei Programmstart sei R1 = 1 und R2 = 2.

```
SP ← (FFFF)16
R1 ← lsh(R1+1)
push_16(R1)
R1 ← lsh(R2+1)
push_16(R1)
R1 ← R1+R2
push_16(R1)
R2 ← R1+R2
pop_16(R2)
pop_16(R1)
R2 ← lsh(R1+1)
pop_16(R1)
```



Aufgabe 2: Stack – Stackmaschine

Entwickeln Sie ein Programm für eine *Stackmaschine*, das diese Folge berechnet und auf dem Stack ablegt:

$$f_n = f_{n-1} + f_{n-2} \text{ für } n \geq 2 \text{ mit } f_0 = 2 \text{ und } f_1 = 1$$

Für beispielsweise $n = 5$ soll der Stack also folgendermaßen aussehen:

11
7
4
3
1
2

Zur Lösung der Aufgabe steht Ihnen eine Stackmaschine mit einem (unendlich großen) Stack und einem Register R zur Verfügung. Das Register R ist mit 1 initialisiert, der Stack ist zu Beginn leer.

Die Stackmaschine kann folgende Operationen ausführen, wobei diese – mit Ausnahme der Operation *pop* – den Wert in R nicht verändern:

- *pop* entfernt den obersten Wert vom Stack und schreibt ihn in das Register R.
- *push* kopiert den Inhalt von Register R auf die oberste Stackposition.
- *swap* vertauscht die beiden obersten Elemente des Stacks.
- *add* entfernt die beiden obersten Elemente vom Stack und legt stattdessen deren Summe an der obersten Position im Stack ab.
- *goto LABEL* führt einen unbedingten Sprung aus, das Programm wird bei LABEL fortgesetzt.

Schreiben Sie ein Programm, das die gegebene Zahlenfolge für $n = \infty$ löst.

Aufgabe 3: Stack – Funktionsaufrufe

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

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. Befehle, die weder den Stack noch eines der Register ändern, sind dabei nicht einzutragen. Die Adressen (`0:` bis `11:`) der Instruktionen sind jeweils links neben dem Pseudocode angegeben. Die ersten drei Zeilen sind bereits vorausgefüllt.

Hinweis: Möglicherweise werden nicht alle Zeilen der Tabelle benötigt.

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

    function cmp() {
 4:    a = a - 1;
 5:    if a > b then {
 6:       cmp();
       }
 7:    a = a + 1;
 8:    inc_b();
 9:    return;
    }

    function inc_b() {
10:    b = b + 1;
11:    return;
    }
```

Adresse	a	b	Stack-Operation	Stack-Inhalt
0	**2**			
1	**2**	**0**		
2	**2**	**0**	**push(3)**	**3**

Aufgabe 4: Adressierungsverfahren – Speicherzugriffssequenz

Gegeben ist der folgende Ausschnitt aus dem Speicher eines Computers, wobei alle Adressen und Konstanten hexadezimal angeschrieben sind:

Adresse	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Wert	3	A	C	4	0	8	9	2	5	E	6	3	5	7	16	12

Auf diesem Computer wird folgende Befehlsfolge ausgeführt (Speicherzugriffsbefehle siehe Foliensatz 11 – Befehlssatz, Folie 10ff sowie *Einführung in die Technische Informatik*, Seite 154ff):

1	R3	←	$memory$[2]
2	R1	←	A
3	R2	←	$memory$[R3]
4	R3	←	$memory$[(R2)+]
5	R1	←	$memory$[R2+3]
6	$memory$[B]	←	$memory$[R1]
7	$memory$[−(R1)]	←	R2
8	$memory$[(R2)+]	←	$memory$[$memory$[(R3)−]
9	$memory$[$memory$[R2]]	←	R3
10	$memory$[(R2)−]	←	R1
11	$memory$[(R2)+]	←	R1
12	$memory$[$memory$[(R2)+]	←	$memory$[$memory$[R1]]
13	$memory$[$memory$[R2]]	←	R3

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/wr) und die zugehörige Adresse an. Alle Register wurden mit 0 initialisiert, die erste Zeile ist bereits vorausgefüllt.

Hinweis: Erfolgt ein Zugriff auf eine nicht angegebene Speicheradresse, sollten Sie Ihren bisherigen Lösungsweg noch einmal überprüfen!

Befehl	R1	R2	R3	Speicherzugriff(e)
1	**0**	**0**	**C**	**rd(2)**
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				

Aufgabe 5: Pipelining – Leiterplattenbestückung

Bei der Bestückung einer Leiterplatte werden fünf Produktionsschritte wie folgt im Pipeline-Verfahren durchlaufen:

1.) Platzieren: Standardbauteile eine Zeiteinheit (ZE), kundenspezifische Bauteile drei ZE
2.) Flussmittelsprühung: eine ZE
3.) Vorwärmen: Standardgröße eine ZE, Übergröße zwei ZE
4.) Lötwelle: Einseitig (Standard) eine ZE, doppelseitig drei ZE
5.) Kühlung: Standardgröße eine ZE, Übergröße drei ZE

Eine Zeiteinheit entspricht genau einem Takt. Jede Leiterplatte muss jeden Produktionsschritt durchlaufen und die betreffenden Maschinen können nur jeweils eine Leiterplatte aufnehmen.

Nachfolgende Leiterplatten sollen im weiteren Verlauf bestückt werden. Bei nicht explizit beschriebenen Produktionsvarianten wird der Standard gemäß obiger Liste verwendet.

- PCB1: Doppelseitig
- PCB2: Übergröße
- PCB3: Kundenspezifische Bauteile, doppelseitig
- PCB4: Übergröße, doppelseitig
- PCB5: Kundenspezifische Bauteile

Gehen Sie bei den nachfolgenden Unteraufgaben immer davon aus, dass die Pipeline anfangs leer ist und somit die erste Leiterplatte ohne Verzögerung bearbeitet werden kann.

a) Wie lange dauert die Bestückung einer einzelnen Leiterplatte vom Typ PCB3?

b) Wie lange dauert es, zehn Leiterplatten vom Typ PCB3 in Folge zu bestücken?

c) Wie lange dauert die Bestückung von drei Leiterplatten in der Reihenfolge PCB3–PCB5–PCB4?

d) Kann die Gesamtdauer aus Teilaufgabe c) durch Umordnung der Reihenfolge verkürzt werden?

e) Angenommen, Sie können wahlweise einen der folgenden Arbeitsschritte verkürzen:

A: Platzieren von kundenspezifischen Bauteilen nur mehr eine statt drei ZE.
B: Lötwelle bei doppelseitigen Platinen nur mehr zwei statt drei ZE.

Welche dieser Verbesserungen bringt mehr Zeitgewinn für die Herstellung von PCB3–PCB4–PCB3–PCB2 ?

Platz für Notizen:

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 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. Andere Arten von *Data Hazards* müssen hier nicht berücksichtigt werden. 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   R4, R5, R4   # Summe R5 + R4 nach R4
PUSH  R4           # R4 auf Stack ablegen
OR    R2, R3, R5   # ODER-Verknüpfung von R3 und R5 nach R2
AND   R6, R4, R5   # UND-Verknüpfung von R4 und R5 nach R6
DIV   R1, R2, R3   # Division von R2 durch R3 nach R1
POP   R3           # oberstes Element vom Stack in R3
```

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!

`ADD`	`PUSH`	`ADD`	`ADD`
`OR`	`ADD`	`AND`	`OR`
`PUSH`	`AND`	`OR`	`PUSH`
`AND`	`OR`	`DIV`	`DIV`
`POP`	`DIV`	`POP`	`AND`
`DIV`	`POP`	`PUSH`	`POP`
❍ 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 n=3 goto L2]* wird der Sprung zur Instruktion mit der Marke *L2* erst beim dritten Mal ausgeführt. Für die Sprungvorhersage werden Sprungbefehle in Stufe F bereits erkannt und im darauffolgenden Zyklus schon der nächste Befehl von der (angenommenen) Zieladresse geladen.

Beide Prozessoren arbeiten mit dynamischer Sprungvorhersage: Wird eine Sprungbedingung erstmals erreicht, wird der Sprung als 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. Der *stall* wird durch Klammerung der Instruktion dargestellt, vgl. *(i3).* 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 12 – 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 n=3 goto L2]
     i2 [goto L1]
L2:  i3
```

Zeit ↓	F	D/E	M/S
1	***i0***		
2	***i1***	***i0***	
3	***(i3)***	***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	***(i3)***	***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?