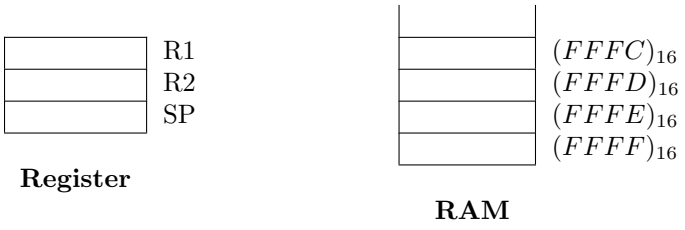
Aufgabe 1: Stack – Funktionsweise eines Stacks

Erläutern Sie die Funktionsweise eines Stacks bzw. Kellerspeichers anhand des folgenden Assembler-Pseudocodes. Tragen Sie die nach Ablauf der Befehlssequenz resultierenden Werte entsprechend ein (vgl. Foliensatz 13 - Befehlssatz, Folie 11ff).

Hinweise: Beachten Sie, dass bei der `pop_16()`-Operation der Speicherinhalt unverändert bleibt, und nur der Wert des Stackpointers angepasst wird. Bei der `push_16()`-Operation wird der Inhalt des angegebenen Registers am Stack abgelegt und der Wert des Stackpointers angepasst. Der Inhalt des Registers bleibt dabei unverändert.

```
SP ← (FFFF)16
R1 ← 0
push_16(R1)
R1 ← 4
push_16(R1)
R1 ← (R1+R1)
push_16(R1)
R1 ← (R1+R1)
push_16(R1)
pop_16(R2)
R2 ← (R1+R2)
pop_16(R1)
R2 ← (R2-R1)
pop_16(R1)
R2 ← (R1+R2)
R2 ← (R1+R2)
```

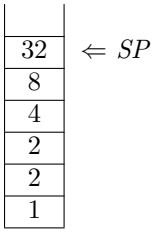


Aufgabe 2: Stack – Stackmaschine

Entwickeln Sie ein Programm für eine *Stackmaschine*, das die angegebene Folge berechnet und auf dem Stack ablegt. Dabei soll die zuletzt berechnete Zahl an der aktuellen Position des Stackpointers liegen.

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

Für beispielsweise $n = 6$ soll der Stack also folgendermaßen aussehen:



Für die Lösung der Aufgabe steht Ihnen eine sogenannte *Stackmaschine* mit genau einem Register R und einem Stack zur Verfügung. Diese *Stackmaschine* kann folgende Operationen ausführen:

- *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.
- *mult* entfernt die beiden obersten Elemente vom Stack und legt stattdessen deren Produkt an der obersten Position im Stack ab.

Mit Ausnahme der Operation *pop* verändern die Operationen den Wert in R nicht. Der Stackpointer wird zu Beginn automatisch korrekt initialisiert und der Stack ist leer. Außerdem ist das Register R mit 1 vorbelegt.

Entwickeln Sie darauf aufbauend eine Folge der beschriebenen Operationen, mit der Sie die gestellte Aufgabe lösen. Brechen Sie die Weiterentwicklung der Operationsfolge ab, sobald Sie ein wiederkehrendes Muster entdeckt haben, mit dem man – wiederholt fortgesetzt – eine beliebige Anzahl von Werten dieser Folge auf dem Stack ablegen kann.

Aufgabe 3: Stack - Funktionsaufrufe

Gegeben ist folgendes Programm in Pseudocode Notation, wobei alle Variablen global deklariert sind.

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 den Stackinhalt *nach* Durchführen der jeweiligen Stackoperation ein.

Hinweis: Die Rücksprungadresse der main()-Funktion müssen Sie nicht beachten.

```
    int main() {
 1:   f();
 2:   u();
 3:   return 0;
    }

    p() {
 5:    b= a+3
 6:    return;
    }

    f() {
 7:    a= r*r;
 8:    p();
 9:    F= b;
10:    return;
    }

    u() {
11:    a= r*3;
12:    p();
13:    U= b;
14:    return;
    }
```

Adresse	Stack-Operation	Stack-Inhalt
1	push(2)	2

Aufgabe 4: Adressierungsverfahren – Speicherzugriffssequenz

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

Adresse	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Wert	6	10	255	21	49	22	0	0	12	127	99	78	25	11	10	16

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

1	$R2$	$\leftarrow$	$memory[0]$
2	$memory[14]$	$\leftarrow$	$R2$
3	$R0$	$\leftarrow$	4
4	$R1$	$\leftarrow$	$memory[(R0)+]$
5	$R2$	$\leftarrow$	$memory[(R0)+]$
6	$R3$	$\leftarrow$	$memory[(R0)+]$
7	$R5$	$\leftarrow$	11
8	$memory[-(R5)]$	$\leftarrow$	$R0$
9	$memory[9]$	$\leftarrow$	$R2$
10	$R5$	$\leftarrow$	$R5 + 1$
11	$R2$	$\leftarrow$	$memory[R5 + 4]$
12	$R6$	$\leftarrow$	$memory[-(R5)]$
13	$R4$	$\leftarrow$	$memory[R5 - 2]$
14	$memory[(R6)]$	$\leftarrow$	$R2$

Tragen Sie den Inhalt der Register nach jedem Befehl in die folgende Tabelle ein. Geben Sie weiters die Adresse an, falls ein Speicherzugriff erfolgt ist.
Hinweis: Gehen Sie davon aus, dass alle Register vor Ablauf der Befehlsfolge mit 0 initialisiert wurden.

Befehl	R0	R1	R2	R3	R4	R5	R6	Speicherzugriff Adresse
1			6					0
2								
3								
4								
5								
6								
7								
8								
9								
10								
11								
12								
13								
14								

Aufgabe 5: Pipeline – Die Autofabrik

In einer Autofabrik werden Autos am Fließband hergestellt. Dabei durchläuft jedes Auto in der folgenden Reihenfolge die 5 Montagestationen: Karosserie (K), Getriebe (G), Motor (M), Elektronik (E) und Inneneinrichtung (I).

Es werden in der Fabrik verschiedene Ausstattungsvarianten auf dem selben Fließband erzeugt. Die Ausstattungsvarianten unterscheiden sich in der Montagezeit wie folgt:

- Für ein Auto mit Standardausstattung benötigt jede Montagestation eine Zeiteinheit.
- Ausstattungsvariante Komfort: Die Montagestation Inneneinrichtung benötigt 3 Zeiteinheiten, alle anderen Stationen 1 Zeiteinheit.
- Ausstattungsvariante Sport: Motor und Getriebe benötigen bei der Montage jeweils 2 Zeiteinheiten, alle anderen Stationen 1 Zeiteinheit.

Pro Montagestation wird immer nur ein Auto bearbeitet. Wird eine Station frei, rückt das Auto von der vorhergehenden Station nach, falls die Bearbeitung dort abgeschloßen ist.

a) Zeichnen Sie den Ablauf am Fließband in einer Arbeitsschicht, wenn Autos in folgender Reihenfolge produziert werden und das Fließband für die Produktion hochgefahren und danach wieder heruntergefahren werden muss:

Standard - Standard - Komfort - Standard - Sport - Standard

b) Wieviele Autos können in der angegebenen Reihenfolge pro Zeiteinheit produziert werden?

c) Wieviele Autos könnten maximal in der angegebenen Reihenfolge pro Zeiteinheit produziert werden, wenn bei einer 24 Stunden Produktion das Hochfahren und Herunterfahren des Fließbandes entfallen würde?

Aufgabe 6: Pipeline – Performanceverbesserung

Ein Prozessor besitzt eine fünfstufige Pipeline: *Fetch*(F), *Decode*(D), *Execute 1*(E1), *Execute 2*(E2) und *Store Result*(S).

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:

Instruktionsart	Fetch	Decode	Execute 1	Execute 2	Store	Summe
i1	200 ns	100 ns	100 ns	50ns	100 ns	550 ns
i2	200 ns	100 ns	100 ns	100ns	200 ns	700 ns
i3	200 ns	100 ns	100 ns	50ns	300 ns	750 ns

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. Die Verarbeitungsstufen, die eine Instruktion durchläuft, sollen dabei immer so aufeinanderfolgen, dass die nachfolgende Verarbeitungsstufe sofort beginnt, nachdem eine Stufe abgeschlossen ist (anders ausgedrückt: Es gibt keine inaktive Zeit zwischen den Verarbeitungsstufen).

b) Der in Teilbeispiel a) verwendete Prozessor soll auf Pipelineverarbeitung umgestellt werden. Aus Kostengründen sollen Sie aber die 5 Verarbeitungsstufen unverändert weiterverwenden (d.h. die Zeit, in der eine Instruktion die Verarbeitungsstufe durchläuft, entspricht der Tabelle). Wie groß wählen Sie unter dieser Voraussetzung die Taktzykluszeit der Pipeline?

c) Berechnen Sie den Geschwindigkeitsgewinn, den der Prozessor durch die Pipeline erzielt. Vergleichen Sie dazu die Taktzykluszeit der parallelen Instruktionsausführung bei gefüllter Pipeline im Vergleich zur Taktzykluszeit der sequenziellen Ausführung der Instruktionen.

Hinweis: Der Einfachheit halber sollen alle Effekte, die den Ablauf der Parallelverarbeitung verzögern könnten, unberücksichtigt bleiben.

d) Zeichnen Sie den genauen zeitlichen Ablauf der Instruktionsausführung innerhalb der Pipelinestufen für alle drei Instruktionstypen.

Stellen Sie dar, wieviel Zeit die jeweilige Instruktion innerhalb jeder Pipelinestufe tatsächlich zur Abarbeitung des entsprechenden Verarbeitungsschrittes braucht und wieviel Zeit ungenützt bleibt.

Was fällt Ihnen dabei auf?

e) Ihr Entwicklungschef hat endlich ein Einsehen und finanziert entweder die Modifikation *einer* Verarbeitungsstufe oder die Verbesserung *eines* Instruktionstyps.

Welche Stufe oder welcher Instruktionstyp bietet sich dafür an und welche Möglichkeiten haben Sie dabei um Ihre Pipeline weiter zu beschleunigen?

Aufgabe 7: Pipelining - RAW-Hazard

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

Bedingt durch die Pipelinestruktur kann es zu *RAW (Read After Write) Data Hazards* kommen, welche durch verzögerte Ausführung des abhängigen Befehls (*stall*) vermieden werden. Dabei wird eine Instruktion erst dann in Stufe D verarbeitet, wenn die abhängige Instruktion Stufe S abgeschlossen hat.

Auf diesem Prozessor wird folgendes Programm ausgeführt:

```
MULT  R4, R4, R4      # R4 quadrieren, Resultat in R4
SLL   R6, R5, 1       # Shift left von R5 um eine Stelle, Resultat in R6
ADD   R2, R3, R4      # R3 und R4 addieren, Resultat in R2
PUSH  R2              # R2 auf dem Stack speichern
DIV   R1, R6, 2       # R6 durch 2 dividieren, Resultat in R1
SUB   R8, R2, R1      # R1 von R2 subtrahieren (R2-R1), Resultat in R8
POP   R9              # R9 vom Stack laden
```

a) Zeichnen Sie die Belegung der Pipeline für das gegebene Programm unter der Annahme, 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				
15				

b) Optimieren Sie das gegebene Programm durch Umordnen der Instruktionen so, dass Verzögerungen, die durch die RAW Data Hazards bei der Ausführung entstehen, soweit wie möglich vermieden werden. Die Funktionalität muss dabei erhalten bleiben!

Aufgabe 8: Pipelining – Control Hazards & Branch Prediction

Eine Teilsequenz eines Programmes umfasst fünf bedingte Sprünge. Folgende Liste gibt die beobachteten Sprungfolgen dieser Sprünge nach einem Programmdurchlauf an (T = branch taken = Sprung ausgeführt, N = branch not taken = Sprung nicht ausgeführt):

Sprung 1: N-N-N-N
Sprung 2: T-T-T-T
Sprung 3: N-T-N-T-T-T-N
Sprung 4: T-T-N-T-T
Sprung 5: T-N-T-N-T-N-T-T-T-N

Untersuchen Sie die nachfolgend beschriebenen *Branch Prediction*-Strategien mit den oben angegebenen Sprungfolgen:

a) Statische Sprungvorhersage – Sprung wird immer als *branch not taken* angenommen.

b) Statische Sprungvorhersage – Sprung wird immer als *branch taken* angenommen.

c) Dynamische Sprungvorhersage

- Wird eine bedingte Sprunginstruktion erstmals ausgeführt, wird der Sprung immer als not-taken angenommen.
- Im Wiederholungsfall wird stets angenommen, dass der Sprung genauso wie beim letzten Mal ausgeführt wird.

Geben Sie für die einzelnen Sprungfolgen an, welcher Sprung bei der jeweiligen Vorhersagestrategie richtig vorhergesagt wurde. Berechnen Sie weiters die Vorhersagegenauigkeit für die Summe aller Sprünge in Prozent (= [Anzahl aller richtig vorausgesagten Sprünge]/[Anzahl aller Sprünge]).

Aufgabe 9: Cache – Direct Mapped Cache

Gegeben ist ein *Direct Mapped Cache* mit 8 Blöcken. Jeder Block kann jeweils 4 Datenwörter aufnehmen. Es wird angenommen, dass ein Datenwort die kleinste adressierbare Einheit ist.

Tragen Sie jeweils in der Tabelle ein, ob der Zugriff auf die angegebene Speicheradresse ein *hit* oder ein *miss* ist, wobei angenommen wird, dass der Cache am Beginn leer ist. Geben Sie bei einem *miss* an, welche Daten in den Cache-Block geladen werden. Verwenden Sie dazu die symbolische Notation *mem[X...Y]*, wobei *X...Y* den Adressbereich angibt, von dem die Daten in den Cache geladen werden.

Adresse	hit/miss	Cache-Block	Inhalt
16	miss	4	*mem[16...19]*
60	miss	7	*mem[60...63]*
1			
34			
18			
59			
2			
3			
35			
19			
20			
16			
56			
57			
21			
5			
4			
58			
36			
4			

Aufgabe 10: Cache – Set Associative Cache

Für einen anfangs leeren Cache ist eine Sequenz von Adresszugriffen gegeben. Der Cache ist ein *4-way set associative* Cache mit 2 Sets zu je 4 Blöcken. Pro Block können 16 Datenworte gespeichert werden. Das Ersetzen des Cache-Inhalts im Konfliktfall erfolgt durch eine *First In-First Out* Ersetzungsstrategie. Das bedeutet, dass der Block, der sich am längsten im jeweiligen Cache-Set befindet, überschrieben wird.

Tragen Sie in die nachfolgende Tabelle ein, ob der jeweilige Speicherzugriff ein *hit* oder ein *miss* ist. Geben Sie an, welcher Cache-Block dabei angesprochen wird. Falls der Zugriff ein *miss* ist, tragen Sie den Inhalt ein, der in den Cache-Block geladen wird. Verwenden Sie dazu die symbolische Notation *mem[X...Y]*, wobei *X...Y* den Adressbereich angibt, von dem die Daten in den Cache geladen werden.

Hinweis: Bitte nummerieren Sie die beiden Cache-Sets mit 0 und 1, sowie die Cache-Blöcke innerhalb jedes Sets jeweils mit 0 . . . 3.

Adresse	hit miss	Cache Set	Block in Set	Inhalt
129	miss	0	0	mem[128...143]
93				
47				
6				
81				
231				
33				
107				
7				
129				
2				
64				
83				
6				
110				