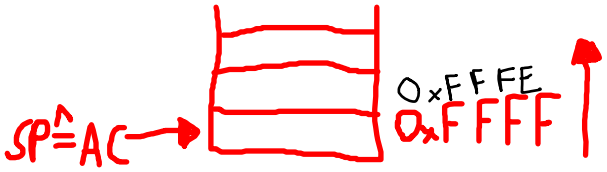
2019W

5. *( _____ / 17 Punkte)* Befehlssatz, Stack

(a) Nachfolgend sind eine Reihe von Maschinenbefehlen gegeben. Übersetzen Sie diese Maschinenbefehle jeweils in funktional äquivalenten, gültigen Micro16-Code. Verwenden Sie hierbei das Micro16 Register `AC` als *Stackpointer*. Der Stack befindet sich am Ende des Speichers, d.h. bei leerem Stack zeigt der Stackpointer auf die Adresse 0xFFFF. Falls temporäre Hilfsregister benötigt werden, verwenden Sie das Register `PC`.

*Hinweis:* Für die Umsetzung eines Maschinenbefehls werden üblicherweise mehrere Micro16-Instruktionen benötigt.

| | | |
|---|---|---|
| 1 | Zuerst R1 um 1 dekrementieren<br>Den Wert auslesen<br>$memory[0x1] \leftarrow memory[-(R1)]$<br>Den gelesenen Wert an die Stelle 1 schreiben | R1 <- R1 + -1<br>MAR <- R1; rd;<br>rd;<br>MAR <- 1; wr;<br>wr;<br>Hier wird der Wert an der Stelle (R1-1) gelesen und an die Stelle 1 geschrieben. Der Wert wird in das MBR gelesen, daher braucht man nur den Wert für das MAR ändern und schreiben, da der Wert noch im MBR ist und der Wert aus dem MBR genommen wird. |
| 2 | $memory[R1] \leftarrow memory[memory[0x7FFF]]$ | RC <- rsh(-1) Dies wird benötigt, um auf den Wert 0x7FFF zu kommen.<br>MAR <- PC; rd;<br>rd; Lesen des Wertes an der Stelle 0x7FFF<br>PC <- MBR; Den Wert in das PC-Register schreiben<br>MAR <- PC; rd; Lesen des nächsten Wertes<br>rd;<br>MAR <- R1; wr; Schreiben des Wertes aus dem MBR an die Stelle R1.<br>wr; |
| 3 | Element vom Stack nehmen<br>$R0 \leftarrow pop()$<br>Da der Stackpointer immer auf den freien Platz zeigt, der Pointer um 1 inkrementiert werden, damit es auf die belegte | AC <- AC + 1<br>MAR <- AC; rd;<br>rd;<br>R0 <- MBR; |
| 4 | Immediate Instruktion. Wir müssen uns den Wert 10 = (0xA) selbst zusammenbauen.<br>$R1 \leftarrow R0 + 0xA$ | PC <- lsh(1+1) # PC = 4<br>PC <- lsh(PC+1) # PC = 10<br>R1 <- R0 + PC |
| 5 | $R2 \leftarrow R0 + R1$ So eine Instruktion existiert im Mikro 16 genauso | R2 <- R0 + R1 |
| 6 | Reinschreiben und dann den Pointer um 1 reduzieren<br>$push(R2)$<br>Etwas auf den Stack pushen. | MAR <- AC; MBR <- R2; wr;<br>wr;<br>AC <- AC + -1 |




AC ist unser Stackpointer, daher muss dieser reduziert werden nach der Operation

(b) Wie nennt man die Operation, die in Zeile 1 auf Register R1 angewandt wird?

Pre-Dekrement

(c) Welche Adressierungsart (*Addressing Mode*) kommt in Zeile 1 auf der rechten Seite der Zuweisung zur Anwendung?

Register Indirect Mode

(d) Welche Adressierungsart (*Addressing Mode*) kommt in Zeile 1 auf der linken Seite der Zuweisung zur Anwendung?

Direct Adress Mode

(e) Wie nennt man das Prinzip, nach dem Schreibe- und Lesezugriffe auf einen Stack funktionieren?

LIFO

3. *( _____ / 14 Punkte)* Das nachfolgende Programm aus Maschinenbefehlen führt Speicherzugriffe durch, legt Daten auf den Stack und liest Daten vom Stack. Gegeben ist zusätzlich eine initiale Speicherbelegung, eine initiale Registerbelegung und der initiale Stackinhalt. Registerinhalte und Konstanten sind in hexadezimaler Notation gegeben.

Speicher:

| Adresse | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Initialer Wert | 5 | 1 | B | 5 | 5 | C | A | F | F | C | B | 3 | 4 | 7 | E | 1 |

```
0    R3 ← 17
1    pop(R3)
2    R1 ← memory[R1]
3    R2 ← memory[memory[B]]
4    push(R1)
5    push(R2)
6    push(R3)
7    push(R3)
8    memory[-(R1)] ← memory[F]
9    pop(R3)
10   memory[(R2)+] ← memory[E]
```

Register:

| | |
|---|---|
| 1 | R1 |
| 0 | R2 |
| 3 | R3 |
| *FFFE* | SP |

Speicherbereich des Stacks:

| | |
|---|---|
| ... | |
| *FFFC* | 5 |
| *FFFD* | 2 |
| *FFFE* | 4 |
| *FFFF* | 7 |

(a) Ermitteln Sie den Speicherinhalt, den Inhalt der Register sowie den Stackinhalt **nach Programmausführung**. Tragen Sie die neuen Werte in die jeweilige Tabelle ein. Beim Speicherinhalt genügt es, nur jene Werte einzutragen, die sich vom initialen Wert unterscheiden.

Speicher:

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

Speicherbereich des Stacks:

| | |
|---|---|
| ... | |
| *FFFC* | |
| *FFFD* | 5 |
| *FFFE* | 1 |
| *FFFF* | 7 |

Register:

| | |
|---|---|
| | R1 |
| | R2 |
| | R3 |
| FFFD | SP |

(b) Wie nennt man die Operation, die in Zeile 10 auf Register R2 angewandt wird?

Post-Dekrement

(c) Welche Adressierungsart (*Addressing Mode*) kommt in Zeile 2 auf der rechten Seite der Zuweisung zur Anwendung?

Register Indirect Mode

(d) Welche Adressierungsart (*Addressing Mode*) kommt in Zeile 8 auf der rechten Seite der Zuweisung zur Anwendung?

Direct Addressing Mode

(e) Wie nennt man das Prinzip, nach dem Schreibe- und Lesezugriffe auf einen Stack funktionieren?

LIFO

7. *( _____ / 11 Punkte)* Folgender eingerückter Text ist analog zu der Stack-Aufgabe in Übung 7:

   Gegeben ist folgendes Programm in Pseudocode-Notation. Die Ausführung startet bei `Program Start()` und endet bei `exit`. Die Variable `P` ist global und die Variable `f` ist lokal deklariert. Die Funktion `fact_rec` hat einen Übergabeparameter. Für die Verwaltung von Rücksprungadressen, lokalen Variablen und Übergabeparameter wird ein Stack verwendet. Lokale Variablen werden bei der ersten Verwendung auf den Stack gelegt und beim Verlassen der Funktion wieder vom Stack entfernt. Zuerst werden Rücksprungadressen und erst dann Übergabeparameter auf den Stack gelegt. Übergabeparameter entsprechen lokalen Variablen in der aufgerufenen Funktion.

   *Hinweis:* Bei Veränderung von lokalen Variablen oder Übergabeparameter werden keine Stack-Operationen durchgeführt. Der Inhalt vom RAM, in dem der Stack liegt, ändert sich allerdings.

   *Hinweis:* Möglicherweise werden nicht alle Zeilen der Tabelle benötigt.

Tragen Sie in die Tabelle ein:

- welche Codezeile (**C**, kann `1:` bis `9:` sein) ausgeführt wurde,
- welchen Wert die globale Variable `P` nach Exekution dieser Codezeile hat,
- welche Stack-Operationen (push(<Wert>)/pop()) ausgeführt werden und
- wie der Stack-Inhalt nach der Codezeile aussieht. Der Stack soll nach **rechts** wachsen.

```
     global P;
     function fact_rec(f) {
1:     if f > 1 then {
2:       P = P*f;
3:       f = f-1;
4:       fact_rec(f);
       }
5:     return;
     }

     Program Start() {
6:     P = 1;
7:     f = 3;
8:     fact_rec(f);
9:     exit;
     }
```

RAM

| C | P | Stack-Operationen | Stack-Inhalt (→) |
|---|---|---|---|
| 6 | 1 | | |
| 7 | 1 | push(3) | 3 |
| 8 | 1 | push(9), push(3) | 3,9,3 |
| 1 | 1 | | 3,9,3 |
| 2 | 3 | | |
| 3 | 3 | | 3,9,2 |
| 4 | 3 | push(5), push(2) | 3,9,2,5,2 |
| 1 | 3 | | 3,9,2,5,2 |
| 2 | 6 | | |
| 3 | 6 | | 3,9,2,5,1 |
| 4 | 6 | push(5), push(1) | 3,9,2,5,1,5,1 |
| 1 | 6 | | |
| 5 | 6 | pop(), pop() | 3,9,2,5,1 |
| 5 | 6 | pop(), pop() | 3,9,2 |
| 5 | 6 | pop(), pop() | 3 |
| 9 | 6 | pop() | |
| | | | |

6. *( _____ / 11 Punkte)* Gegeben ist folgendes Programm in Pseudocode-Notation. Die Ausführung startet bei `Program Start()` und endet bei `exit`. Die Variable `sum` ist global und die Variable `n` sind lokal deklariert. Die Funktion `sum_rec` hat einen Übergabeparameter. Für die Verwaltung von Rücksprungadressen und Übergabeparameter wird ein Stack verwendet. Zuerst werden Rücksprungadressen und erst dann Übergabeparameter auf den Stack gelegt.

   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 globalen Variablen und den Inhalt des Stacks nach Durchführung der jeweiligen Instruktion ein. Die Adressen (`0:` bis `8:`) der Instruktionen sind jeweils links neben dem Pseudocode angegeben.

   *Hinweis:* Übergabeparameter entsprechen lokalen Variablen in den aufgerufenen Funktionen und werden also erst am Ende der Funktion vom Stack gelöscht.

   *Hinweis:* Möglicherweise werden nicht alle Zeilen der Tabelle benötigt.

```
      global sum;
      function sum_rec(n) {
0:      if n > 0 then {
1:        sum = sum+n;
2:        n = n-1;
3:        sum_rec(n);
        }
4:      return;
      }

      Program Start() {
5:      sum = 0;
6:      n = 2;
7:      sum_rec(n);
8:      exit;
      }
```

| Adresse | sum | Stack-Operation | Stack-Inhalt |
|---|---|---|---|
| 5 | 0 | | |
| 6 | 0 | push(2) | 2 |
| 7 | 0 | push(8), push(2) | 2, 8, 2 |
| 0 | 0 | | |
| 1 | 2 | | |
| 2 | 2 | | 2, 8, 1 |
| 3 | 2 | push(4), push(1) | 2, 8, 1, 4, 1 |
| 0 | 2 | | |
| 1 | 3 | | |
| 2 | 3 | | 2, 8, 1, 4, 0 |
| 3 | 3 | push(4), push(0) | 2, 8, 1, 4, 0, 4, 0 |
| 0 | 3 | | |
| 4 | 3 | pop(0), pop(4) | 2, 8, 1, 4, 0 |
| 4 | 3 | pop(0), pop(4) | 2, 8, 1 |
| 4 | 3 | pop(1), pop(8) | 2 |
| 8 | 3 | pop(2) | |
| | | | |
| | | | |
| | | | |
| | | | |

2017S

2. *(12 Punkte)* Beantworten Sie folgende Fragen zu Speicherzugriffen und Stacks.

(a) Welche der folgenden Instruktionspaare stellen korrekte Implementierungen der Stack Operationen *push(R)* und *pop(R)* dar? Wenn ein Instruktionspaar *push(R)* und *pop(R)*, in beliebiger Reihenfolge, darstellt, schreiben Sie unter die entsprechende Instruktion *push(R)* bzw. *pop(R)*. Stellt ein Instruktionspaar nicht beides, *push(R)* und *pop(R)*, dar, so streichen Sie das komplette Instruktionspaar durch.

Der Stackpointer zeigt immer auf die naechste freie Adresse.


| | |
|---|---|
| $memory[SP-] \leftarrow R$ | $R \leftarrow memory[+SP]$ |
| push | POP |
| ~~$R \leftarrow memory[SP-]$~~ | ~~$memory[-SP] \leftarrow R$~~ |
| | |
| ~~$R \leftarrow memory[+SP]$~~ | ~~$memory[SP+] \leftarrow R$~~ |
| | |
| ~~$memory[SP+] \leftarrow R$~~ | ~~$R \leftarrow memory[SP-]$~~ |
| Push(R) | |

Kann nicht sein, da beides dekrementiert wird

Kann nicht sein, da beides inkrementiert wird

Kann nicht sein, da beide Operationen post krement Operationen sind.

(b) Gegeben sind nachfolgende Instruktionen. Schreiben Sie jeweils rechts neben der Instruktion die verwendete Adressierungsart hin.

*Hinweis:* In der Vorlesung sind folgende Adressierungsarten betrachtet worden: Direct-Addressing Mode, Immediate Mode, Indirect-Addressing Mode, Register-Indirect Mode und Register Mode.

| | |
|---|---|
| $R1 \leftarrow R2$ | Register-Mode |
| $R3 \leftarrow -1$ | Immediate Mode |
| $R0 \leftarrow memory[0x500]$ | Direct-Adressing Mode |
| $R3 \leftarrow memory[R4]$ | Register-Indirect Mode |
| $R7 \leftarrow memory[memory[0x666]]$ | Indirect-Adressing Mode |

(c) Welche der Abkürzungen *LIFO, LILO, FIFO* und *FILO* beschreibt die Funktionsweise eines Stacks und wofür steht die Abkürzung?

LIFO, FILO

4. *(8 Punkte)* Gegeben ist ein Programm mit Funktionsaufrufen. Die Variable `x` ist global definiert.

   (a) Bei jedem Funktionsaufruf wird die Rücksprungadresse (Zeilennummer) am Stack abgelegt. Der Stackpointer (SP) zeigt zu Beginn auf die Adresse 0xFFFF. Ermitteln Sie den Stackpointer und den Stackinhalt nach Ausführung der Funktion `Program Start()`.

   *Hinweis:* Beachten Sie, dass nicht mehr benötigte Stack-Einträge am Stack bleiben und bei nachfolgenden Stack-Zugriffen überschrieben werden. Weiters, dass Variable `x` global definiert ist.

```
    Program Start() {
0:      x = 0;
1:      u();
2:      v();
3:      exit;
    }

    function v() {
4:      if x > 0 then {
5:          x = x-1;
6:          v(x);
        }
7:      return;
    }

    function u() {
8:      x = 3;
9:      return;
    }
```

Stack

| | |
|---|---|
| 0xFFF9 | |
| 0xFFFA | |
| 0xFFFB | |
| 0xFFFC | 7 |
| 0xFFFD | 7 |
| 0xFFFE | 7 |
| 0xFFFF | 3 2 |

Am Ende des Programms muss der Stack auf der Adresse 0xFFFF stehen, weil ja alles vom Stack heruntergenommen werden muss

Register

| | |
|---|---|
| SP | 0xFFFF |

   (b) Kreuzen Sie an, ob es sich um wahre oder falsche Aussagen handelt.
   *(richtig: +1 Punkte, falsch: -1 Punkte, keine Antwort: 0 Punkte)*

| wahr | falsch | |
|---|---|---|
| ❍ | ✗ | `If`-Anweisungen können bei Prozessoren mit Pipelining zu Structural Hazards führen. Control Hazards wäre die richtige Antwort |
| ❍ | ✗ | Bei einer RISC Prozessorarchitektur stehen tendenziell mehr Maschinenbefehle zur Verfügung als bei einer CISC Prozessorarchitektur. Genau umgekehrt |
| ❍ | ✗ | Eine CISC Prozessorarchitektur arbeitet tendenziell mit einer höheren Taktrate als eine RISC Prozessorarchitektur. Bei einem CISC habe ich eine kompliziertere Hardware. Daher kann ich nicht so einen hohen Takt haben. |

2. *(8 Punkte)* Stack und Register eines Prozessors sind wie folgt initialisiert, wobei alle Werte hexadezimal angegeben sind:

| | Stack |
|---|---|
| ... | |
| *FFFC* | |
| *FFFD* | 8 |
| *FFFE* | 5 |
| *FFFF* | 3 |

| Register | |
|---|---|
| 0 | R1 |
| 0 | R2 |
| 0 | R3 |
| *FFFC* | SP |

(a) Auf dem Prozessor werden die unten links angeführten sechs Befehle von oben nach unten ausgeführt. Tragen Sie die Werte auf dem Stack, der Register und des Stackpointers in hexadezimaler Form jeweils nach Ausführung der beiden Instruktionen ein!
Geben Sie in eckigen Klammern [ ] auch jene Daten auf dem Stack an, die durch vorangegangene `pop()`-Befehle bereits ungültig sind, die aber noch nicht überschrieben wurden!

`pop(R1)`
`pop(R2)`

| | Stack |
|---|---|
| ... | |
| *FFFC* | |
| *FFFD* | [8] |
| *FFFE* | [5] ← SP |
| *FFFF* | 3 |

| Register | |
|---|---|
| 8 | R1 |
| 5 | R2 |
| 0 | R3 |
| FFFE | SP |

---

`push(R1)`
`R1 ← R1 + R2`

| | Stack |
|---|---|
| ... | |
| *FFFC* | |
| *FFFD* | [8] ← SP |
| *FFFE* | 8 |
| *FFFF* | 3 |

| Register | |
|---|---|
| D | R1 |
| 5 | R2 |
| 0 | R3 |
| FFFD | SP |

---

`pop(R3)`
`push(R1)`

| | Stack |
|---|---|
| ... | |
| *FFFC* | |
| *FFFD* | [8] ← SP |
| *FFFE* | D |
| *FFFF* | 3 |

| Register | |
|---|---|
| D | R1 |
| 5 | R2 |
| 8 | R3 |
| FFFD | SP |

(b) Kreuzen Sie die korrekte Aussage an!
*(richtig: +2 Punkte, falsch: -2 Punkte, nicht gekreuzt: 0 Punkte)*

- ❍ Die Instruktion R1 ← *memory*[−(SP)] realisiert `pop(R1)`.
- ❍ Die Instruktion R1 ← *memory*[(SP)−] realisiert `pop(R1)`.
- ❍ Die Instruktion R1 ← *memory*[(SP)+] realisiert `pop(R1)`.
- ☒ Die Instruktion R1 ← *memory*[+(SP)] realisiert `pop(R1)`.

push(R1): memory[SP-] ← R1