KONZEPTE SYSTEMNAHER PROGRAMMIERUNG

Technische Hochschule Mittelhessen

Andre Rein

– Ninja Stackmaschine, erste Instruktionen und Speicher –

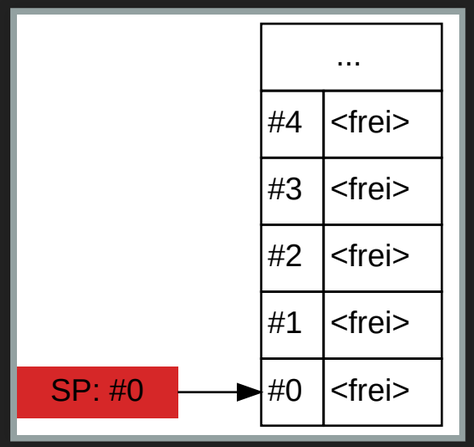
NINJA VM: STACKMASCHINE

- Die Ninja Virtual Machine (njvm) basiert auf den Konzepten einer sog. Stackmaschine. Hierbei erfolgen alle Berechnungen und die vollständige Steuerung unter zu Hilfenahme eines Last-in-First-out (LIFO → Stack) Speichers.
- Das Grundprinzip einer Stackmaschine basiert darauf, dass Operanden einer Berechnung mittels den Operationen `push` auf den Stack gelegt und mittels `pop` vom Stack geholt und entfernt werden können.

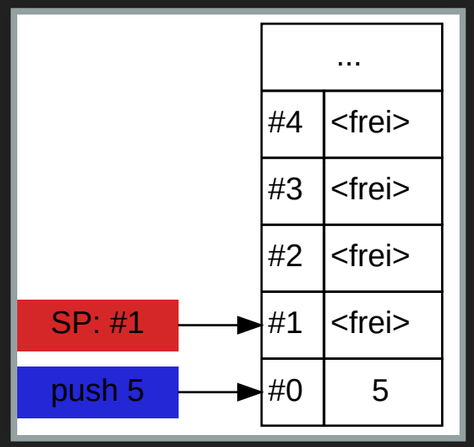
Eine konkrete Berechnung, z.B. eine Addition, holt dann **2** Operanden (`a=pop()`, `b=pop()`) vom Stack, führt die Berechnung aus (`c=a+b`) und speichert das Ergebnis wieder auf dem Stack (`push c`).

NINJA VM: STACKMASCHINE

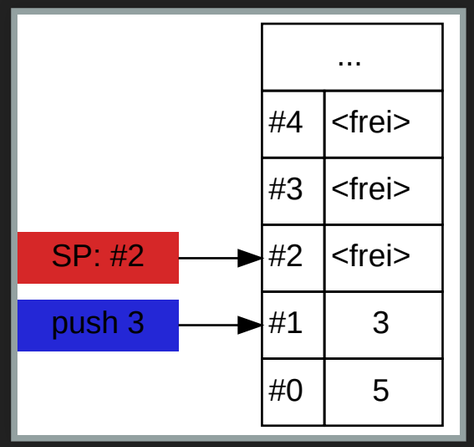
- Mit der Operation `push <Wert>` werden Elemente auf die oberste Position des Stack gelegt, das ist die Position auf die der sog. Stackpointer `SP` aktuell zeigt. Anschließend wird der `SP` um den Wert `1` inkrementiert.
- Mit den Operationen `pop` wird das oberste Element auf den Stack, also an der Position auf die der Stackpointer `SP` aktuell zeigt, herunter genommen. Anschließend wird der `SP` um den Wert `1` dekrementiert.



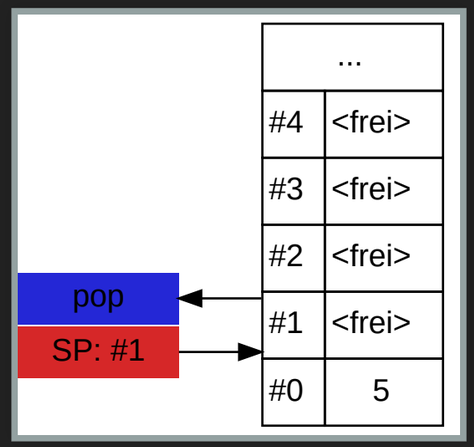

```
push 5
push 3
pop
```




```
push 5 <--
push 3
pop
```




```
push 5
push 3 <--
pop
```




```
push 5
push 3
pop    <--
```

NINJA VM: STACKMASCHINE BEISPIELCODE

Datei stack1.c

```c
#include <stdio.h>
#include <stdlib.h>

#define MAXITEMS 100

int sp=0; // the stackpointer
int stack[MAXITEMS];

void push(int x) {
    printf("-[%4s]-> pushing [%d] onto stack @sp [%d]\n", __func__, x, sp);
    stack[sp]=x;
    printf("-[%4s]-> inc stack pointer [%d -> ", __func__, sp);
    sp++;
    printf("%d]\n", sp);
}

int pop(void) {
    printf("-[%4s]-> dec stack pointer [%d -> ", __func__, sp);
    sp--;
    printf("%d]\n", sp);
    int tmp = stack[sp];
    printf("-[%4s]-> popping [%d] from stack @sp [%d]\n", __func__, tmp, sp);
    return tmp;
}

void print_stack(void) {
    printf("\n     Stack\n");
    printf(".-------+--------.\n");
    for (int i=sp; i>=0; i--) {
        if (i==sp)
            printf("|sp->%3d| <empty>|\n", i);
        else
            printf("|%7d|  %5d |\n", i, stack[i]);
    }
    printf("'-------+--------'\n\n");
}

int main (int argc, char *argv[]) {
    int value;
    print_stack();
    push(5);
    print_stack();
    push(3);
    print_stack();
    value=pop();
    print_stack();
    return 0;
}
```

Beispielausgabe

```
$ gcc -o stack1 stack1.c && ./stack1

     Stack
.-------+--------.
|sp->  0| <empty>|
'-------+--------'

-[push]-> pushing [5] onto stack @sp [0]
-[push]-> inc stack pointer [0 -> 1]

     Stack
.-------+--------.
|sp->  1| <empty>|
|      0|      5 |
'-------+--------'

-[push]-> pushing [3] onto stack @sp [1]
-[push]-> inc stack pointer [1 -> 2]

     Stack
.-------+--------.
|sp->  2| <empty>|
|      1|      3 |
|      0|      5 |
'-------+--------'

-[ pop]-> dec stack pointer [2 -> 1]
-[ pop]-> popping [3] from stack @sp [1]

     Stack
.-------+--------.
|sp->  1| <empty>|
|      0|      5 |
'-------+--------'
```

INSTRUKTIONEN DER VM

- Die Ninja VM besteht im groben aus dem Stack-Speicher und **Instruktionen** die den Stack nutzen um bestimmte Operationen auszuführen
  - **Beispiel**: Arithmetische Operationen wie *Addition*, *Subtraktion*, …
- Instruktionen kennen wir typischerweise aus dem CPU-Umfeld und der Rechnerarchitektur:
  - Jede CPU verfügt über einen sog. **Befehlssatz** (engl. instruction set), also eine Anzahl an **Maschinenbefehlen** (Instruktionen) die durch die CPU **ausgführt** werden können
- Dabei müssen Speicher und Befehlssatz aufeinander abgestimmt sein
  - Für die Ninja VM bedeutet das — **Der Stack ist die zentrale Speichereinheit, die von allen Instruktionen verwendet wird, um Operationen auszuführen**
- D.h. jede Operation lässt sich beschreiben, indem man die Modifikationen darstellt, die am Stack vorgenommen werden
  - Beispiel: `pushc <const>` → `... -> ... value`

INSTRUKTIONEN DER VM: ÜBERSICHT

Instruktion	Opcode	Stack Layout
`halt`	`0`	`... -> ...`
`pushc <const>`	`1`	`... -> ... value`
`add`	`2`	`... n1 n2 -> ... n1+n2`
`sub`	`3`	`... n1 n2 -> ... n1-n2`
`mul`	`4`	`... n1 n2 -> ... n1*n2`
`div`	`5`	`... n1 n2 -> ... n1/n2`
`mod`	`6`	`... n1 n2 -> ... n1%n2`
`rdint`	`7`	`... -> ... value`
`wrint`	`8`	`... value -> ...`
`rdchr`	`9`	`... -> ... value`
`wrchr`	`10`	`... value -> ...`

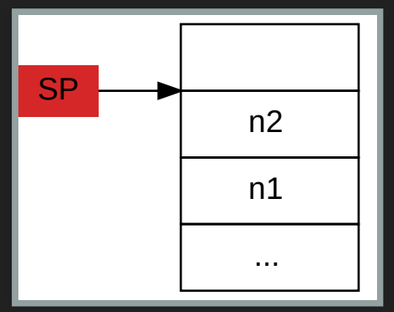
Diese Instruktionen sind relevant für die Aufgabe 1 und sind auf dem Aufgabenblatt angegeben.

INSTRUKTIONEN DER VM: BEISPIEL ADDITION

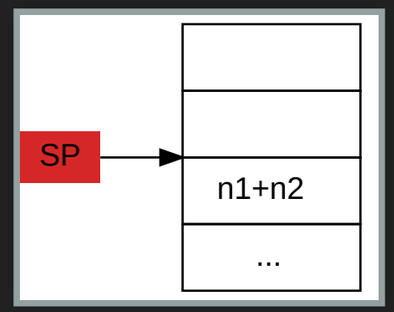
Instruktion	Opcode	Stack Layout
`add`	`2`	`... n1 n2 -> ... n1+n2`

- Die Addition wird von der Instruktion `add` implementiert, die Veränderungen auf dem Stack sind:

Stack vor der Operation: `... n1 n2`



Stack nach der Operation: `... n1+n2`



`...` gibt hierbei an, dass der Inhalt beliebig ist, er spielt keine Rolle für die Operation!

INSTRUKTIONEN DER VM: ARITHMETISCHE OPERATIONEN

Instruktion	Opcode	Stack Layout
`add`	`2`	`... n1 n2 -> ... n1+n2`
`sub`	`3`	`... n1 n2 -> ... n1-n2`
`mul`	`4`	`... n1 n2 -> ... n1*n2`
`div`	`5`	`... n1 n2 -> ... n1/n2`
`mod`	`6`	`... n1 n2 -> ... n1%n2`

- Alle arithmetischen Operationen verhalten sich bezüglich der Nutzung des Stacks äquivalent; nur das Ergebnis der eigentlichen Operation (`+, -, *, /, %`) unterscheidet sich!

INSTRUKTIONEN DER VM: ANDERE OPERATIONEN

Instruktion	Opcode	Stack Layout	Kurzerklärung
`halt`	`0`	`... -> ...`	Beendet die Ausführung des Programms
`pushc <const>`	`1`	`... -> ... value`	Legt einen Wert an der **obersten** Stackposition ab
`rdint`	`7`	`... -> ... value`	Liest einen **Integer**, speichert ihn an der **obersten** Stackposition ab
`wrint`	`8`	`... value -> ...`	Entfernen und Ausgeben des **Integers**, an **oberster** Stackposition
`rdchr`	`9`	`... -> ... value`	Liest einen **Character**, speichert ihn an der **obersten** Stackposition ab
`wrchr`	`10`	`... value -> ...`	Entfernen und Ausgeben des **Characters**, an **oberster** Stackposition

- Die *anderen* Instruktionen führen Operationen aus, die zum Betrieb der Ninja VM benötigt werden, wie z.B. *Beenden* oder *Eingabe/Ausgabe*
- Später kommen weitere Instruktionen hinzu, die weitere Operationen ermöglichen. Z.B. *Boolesche Vergleichsoperationen*, *Funktionsaufrufe*, *Sprünge* , …

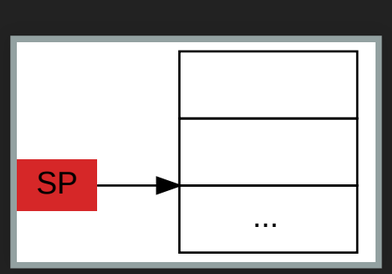
⚠	**Achtung**: Das Einlesen eines Characters mit `rdchr`, z.B. `A`, liest hierbei den (Integer-)Wert `65` ein und speichert ihn auf dem Stack ab. Bei der Ausgabe dieses Wertes mit `wrchr` wird abermals ein `A` ausgegeben *(und nicht etwa der Wert `65`; dieser würde bei einer Ausgabe mit `wrint` ausgegeben)*.

INSTRUKTIONEN DER VM: BEISPIEL PUSHC

Instruktion	Opcode	Stack Layout
`pushc <const>`	`1`	`... -> ... value`

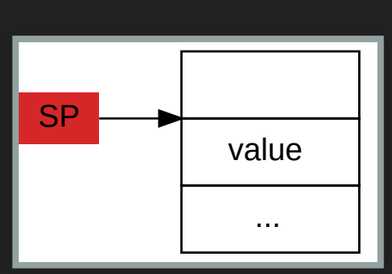
Stack vor der Operation:

`...`



Stack nach der Operation:

`... value`



`...` gibt hierbei an, dass der Inhalt beliebig ist, er spielt keine Rolle für die Operation!

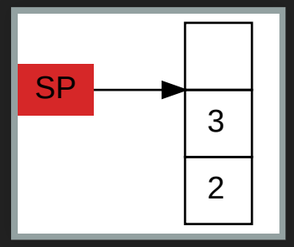
INSTRUKTIONEN DER VM: BEISPIEL $2 * 3 + 5$

- Die Berechnung des Ausdrucks $2 * 3 + 5$ und der Ausgabe des Ergebnisses kann in Ninja mit folgenden Instruktionen erfolgen:

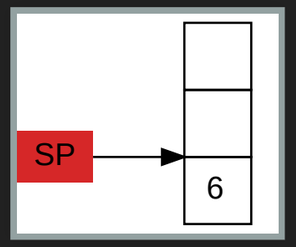
Datei ninjatest1.asm

```
pushc 2
pushc 3
mul
pushc 5
add
wrint
```

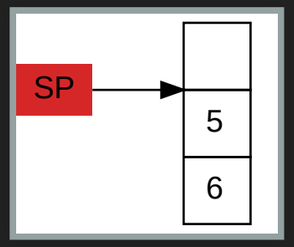
INSTRUKTIONEN DER VM: BEISPIEL $2 * 3 + 5$



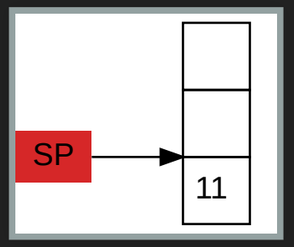

```
pushc 2
pushc 3 <-
mul
pushc 5
add
wrint
```



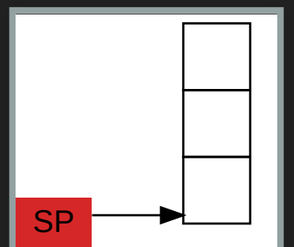

```
pushc 2
pushc 3
mul     <-
pushc 5
add
wrint
```




```
pushc 2
pushc 3
mul
pushc 5 <-
add
wrint
```




```
pushc 2
pushc 3
mul
pushc 5
add     <-
wrint
```




```
pushc 2
pushc 3
mul
pushc 5
add
wrint   <- Ausgabe 1
```

KODIERUNG DER NINJA VM INSTRUKTIONEN

- Die Instruktionen der Ninja VM sind auf eine spezielle Art und Weise **kodiert**
- Hierdurch ist es möglich sowohl die Instruktion selbst (mittels dem sog. **Opcode**), als auch Nutzdaten (wie z.B. Integerwerte) in nur einem Befehl zu übertragen
  - Um diese Kodierung kümmert sich (später) der Ninja Assembler, der aus gegebenem Ninja Assembler Code → Ninja Byte Code erzeugt
- Die Ninja VM erhält nun die kodierten Instruktionen als Bytecode und verarbeitet die einzelnen Instruktionen anhand der getroffenen Spezifikationen
  - Also: Manipulation des Stacks und Ausführung der angegebenen Operationen

KODIERUNG DER NINJA VM INSTRUKTIONEN

- Jede Ninja-Instruktion wird als ein 32 Bit Wert repräsentiert
  - Hierbei handelt es sich um einen Wert **ohne Vorzeichen**, also z.B. um einen Wert vom Typ `unsigned int`

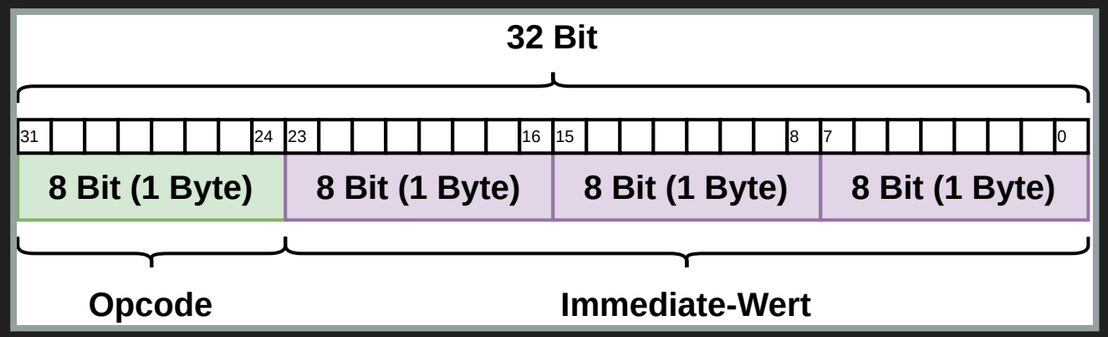
```
unsigned int instruction;
```

- Alternativ kann der Typ `uint32_t` aus dem Header `stdint.h` verwendet werden, hier ist sichergestellt das es sich tatsächlich um einen 32-Bit Wert handelt, `unsinged int` ist typischerweise zwar 32 Bit, aber dies ist in C nicht genau spezifiziert.

```
#include <stdint.h>
uint32_t instruction;
```

KODIERUNG DER NINJA VM INSTRUKTIONEN

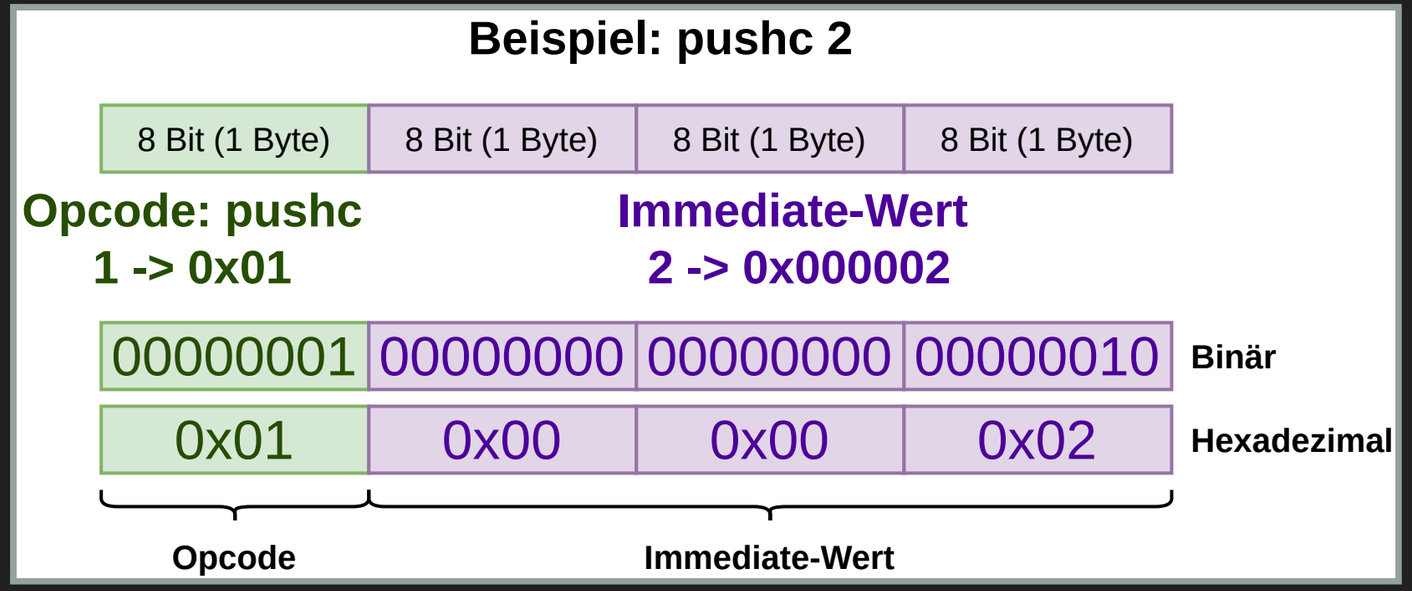
- Der verwendete 32 Bit Wert ist in 2 Bereiche aufgeteilt
  - **Opcode** → Die höchsten 8 Bit
  - **Immediate-Wert** → Die 24 niederwertigen Bits



- Insgesamt gibt es maximal 256 **Opcodes**, d.h. die Ninja VM kann auch maximal nur 256 Instruktionen haben — *Was mehr als ausreichend ist!*
- Der Immediate-Wert kann im Wertebereich insgesamt $2^{24} = 16777216$ Werte repräsentieren
  - Wertebereich `unsigned`: $0 \ldots 16777215$
  - Wertebereich `signed`: $-8388608 \ldots 0 \ldots 8388607$
- **ACHTUNG:** Die Immediate-Werte werden bei uns `signed`, also vorzeichenbehaftet, sein

KODIERUNG BEISPIEL: PUSHC 2

- Die Instruktion `pushc 2` gliedert sich also auf in den
  - **Opcode**: `1` (für `pushc`) und den
  - **Immediate-Wert**: `2`




```c
unsigned int instruction = 0x01000002;
```

KODIERUNG DER NINJA VM INSTRUKTIONEN

Bei Operationen die keine Immediate-Wert verwenden (die meisten verwenden keinen), wird nur der Opcode-Anteil der Instruktion verwendet und der Immediate-Wert bleibt auf `0` (24 Bit).

- Einige Beispiele:
  - Instruktion `mul` (Opcode: 4) → `0x04000000`,
  - Instruktion `wrchr` (Opcode: 10) → `0x0A000000`,
  - ...

KODIERUNG VON OPCODES

- Die Verwendung der Opcode-Werte, also der Zahlen, ist für Menschen sicherlich nicht optimal
- Anstelle der Zahlen können wir nun einfach den Präprozessor verwenden, um uns das **Lesen** des Programmcodes zu vereinfachen.

```
#define HALT   0
#define PUSHC  1
...
#define MUL    4
...
#define WRCHR 10
...
```

```
#define MUL 4
unsigned int instruction = MUL; // <- 0x00000004 falsch!
```

ist jedoch falsch, da wir den Opcode in den 8 höchstwertigen Bits benötigen. Hierzu kann man eine sog. Bit-Shift Operation verwenden.

```
#define MUL 4
unsigned int instruction = MUL << 24; // <- 0x04000000
```

KODIERUNG VON OPCODES MIT IMMEDIATE-WERTEN

- Falls die Instruktion einen Immediate-Wert enthält, wie z.b. `pushc 2`, erfolgt auch ein Bitshift um 24 Positionen für den Opcode. Der Immediate-Wert wird dann mit Hilfe der bitweisen *Oder*-Operation mit der Instruktion verodert.

```
#define PUSHC
unsigned int instruction = (PUSHC << 24) | 2; // <- 0x01000002
```

Datei shift.c

```
#include <stdio.h>
#include <string.h>

#define PUSHC 1

int main(int argc, char *argv[]){
    unsigned int instruction;
    for (int i = 0; i < 8; ++i) {
        instruction = (PUSHC << 24) | i; // -> pushc i
        printf("[PUSHC] (%d << 24 | %2d) = 0x%08x\n",
                PUSHC, i, instruction);
    }
    return 0;
}
```

Ausgabe

```
$ gcc -o shift shift.c && ./shift
[PUSHC] (1 << 24 |  0) = 0x01000000
[PUSHC] (1 << 24 |  1) = 0x01000001
[PUSHC] (1 << 24 |  2) = 0x01000002
[PUSHC] (1 << 24 |  3) = 0x01000003
[PUSHC] (1 << 24 |  4) = 0x01000004
[PUSHC] (1 << 24 |  5) = 0x01000005
[PUSHC] (1 << 24 |  6) = 0x01000006
[PUSHC] (1 << 24 |  7) = 0x01000007
```

NEGATIVE IMMEDIATE-WERTE

- Immediate-Werte können sowohl **positiv**, als auch **negativ** sein.
- Hierbei tritt allerdings ein Problem mit der Kodierung auf, denn negative Werte sind in C im 2er-Komplement kodiert.
  - `-1` entspricht `0xffffffff`

Datei shift2.c

```c
#include <stdio.h>
#include <string.h>

#define PUSHC 1

int main(int argc, char *argv[]){
    unsigned int instruction;
    for (int i = -4; i < 4; ++i) {
        instruction = (PUSHC << 24) | i; // -> pushc i
        printf("[PUSHC] (%d << 24 | %2d) = 0x%08x\n",
                PUSHC, i, instruction);
    }
    return 0;
}
```

Ausgabe

```
$ gcc -o shift2 shift2.c && ./shift2
[PUSHC] (1 << 24 | -4) = 0xfffffffc
[PUSHC] (1 << 24 | -3) = 0xfffffffd
[PUSHC] (1 << 24 | -2) = 0xfffffffe
[PUSHC] (1 << 24 | -1) = 0xffffffff
[PUSHC] (1 << 24 |  0) = 0x01000000
[PUSHC] (1 << 24 |  1) = 0x01000001
[PUSHC] (1 << 24 |  2) = 0x01000002
[PUSHC] (1 << 24 |  3) = 0x01000003
```

Wie man sieht gehen hierbei die Opcodes verloren!

NEGATIVE IMMEDIATE-WERTE

- Bei den dargestellten negativen Zahlen, waren alle 32 Bits belegt
  - Aus diesem Grund wurden die Opcodes überschrieben
- Unsere Kodierung verwendet jedoch nur die 24 niederwertigen Bits
  - Wir müssen also die 8 höchstwertigen Bits von unseren Immediate-Werten *abscheiden*
  - Dies erfolgt mit der bitweisen **verundung** mit dem Wert `0x00FFFFFF`

Datei shift3.c

```c
#include <stdio.h>
#include <string.h>

#define IMMEDIATE(x) ((x) & 0x00FFFFFF)
#define PUSHC 1

int main(int argc, char *argv[]){
    unsigned int instruction;
    for (int i = -4; i < 4; ++i) {
        instruction = (PUSHC << 24) | IMMEDIATE(i);
        printf("[PUSHC] (%d << 24 | %2d) = 0x%08x\n",
                PUSHC, i, instruction);
    }
    return 0;
}
```

Ausgabe

```
$ gcc -o shift3 shift3.c && ./shift3
[PUSHC] (1 << 24 | -4) = 0x01fffffc
[PUSHC] (1 << 24 | -3) = 0x01fffffd
[PUSHC] (1 << 24 | -2) = 0x01fffffe
[PUSHC] (1 << 24 | -1) = 0x01ffffff
[PUSHC] (1 << 24 |  0) = 0x01000000
[PUSHC] (1 << 24 |  1) = 0x01000001
[PUSHC] (1 << 24 |  2) = 0x01000002
[PUSHC] (1 << 24 |  3) = 0x01000003
```

Die Ausgabe ist nun korrekt!

INSTRUKTIONSZERLEGUNG: OPCODE UND IMMEDIATE

- Der Assembler übernimmt für uns die Kodierung der Instruktionen wenn der Bytecode erzeugt wird.
  - D.h. in späteren Aufgaben die einen Compiler und Assembler beinhalten, müssen die kodierten Instruktionen wieder in Opcode und Immediate-Wert zerlegt werden.
- Die Zerlegung in den Opcode ist hierbei ein Bitshift nach rechts um 24 Positionen `>> 24`
- Die Zerlegung in den Immediate-Wert erfolgt mit der *Verundung* mit dem Wert `& 0x00FFFFFF`
- Negative Immediate-Wert benötigen abermals eine Sonderbehandlung.

```
#define SIGN_EXTEND(i) ((i) & 0x00800000 ? (i) | 0xFF000000 : (i))
```

- **Makro**: Wenn Bit 23 eine `1` ist, dann handelt es sich um eine negative Zahl, die durch 8 weitere `1`-en in den höchstwertigen Bits erweitert werden muss.

INSTRUKTIONSZERLEGUNG: OPCODE UND IMMEDIATE

Beispiel ohne `SIGN_EXTEND` Makro liefert falsche Ergebnisse bei negativen Immediate-Werten!

Datei shift4.c

```c
#include <stdio.h>
#include <string.h>

#define IMMEDIATE(x) (x & 0x00FFFFFF)
#define PUSHC 1

int main(int argc, char *argv[]){
    unsigned int instruction;
    for (int i = -4; i < 4; ++i) {
        instruction = (PUSHC << 24) | IMMEDIATE(i);

        printf("0x%08x -> Opcode [%d] Immediate [%d]\n",
               instruction, (instruction >> 24),
               (instruction & 0x00FFFFFF));
    }
    return 0;
}
```

Ausgabe

```
$ gcc -o shift4 shift4.c && ./shift4
0x01fffffc -> Opcode [1] Immediate [16777212]
0x01fffffd -> Opcode [1] Immediate [16777213]
0x01fffffe -> Opcode [1] Immediate [16777214]
0x01ffffff -> Opcode [1] Immediate [16777215]
0x01000000 -> Opcode [1] Immediate [0]
0x01000001 -> Opcode [1] Immediate [1]
0x01000002 -> Opcode [1] Immediate [2]
0x01000003 -> Opcode [1] Immediate [3]
```

INSTRUKTIONSZERLEGUNG: OPCODE UND IMMEDIATE

Datei shift5.c

```c
#include <stdio.h>
#include <string.h>

#define IMMEDIATE(x) (x & 0x00FFFFFF)
#define SIGN_EXTEND(i) ((i) & 0x00800000 ? (i) \
        | 0xFF000000 : (i))
#define PUSHC 1


int main(int argc, char *argv[]){
    unsigned int instruction;
    int opcode;
    int immediate;
    for (int i = -4; i < 4; ++i) {
        instruction = (PUSHC << 24) | IMMEDIATE(i);
        opcode = instruction >> 24;
        immediate = SIGN_EXTEND(instruction & 0x00FFFFFF);
        printf("0x%08x -> Opcode [%d] Immediate [%d]\n",
                instruction, opcode, immediate);
    }
    return 0;
}
```

Ausgabe

```
$ gcc -o shift5 shift5.c && ./shift5
0x01fffffc -> Opcode [1] Immediate [-4]
0x01fffffd -> Opcode [1] Immediate [-3]
0x01fffffe -> Opcode [1] Immediate [-2]
0x01ffffff -> Opcode [1] Immediate [-1]
0x01000000 -> Opcode [1] Immediate [0]
0x01000001 -> Opcode [1] Immediate [1]
0x01000002 -> Opcode [1] Immediate [2]
0x01000003 -> Opcode [1] Immediate [3]
```

Die Ausgabe ist nun korrekt!

PROGRAMMSPEICHER UND PROGRAM COUNTER

- Der Programmspeicher ist der Speicher in dem das Programm hinterlegt ist

Beispiel: Programm zur Berechnung und Ausgabe von $2 * 3 + 5$

Programm in Assembler und Bytecode

```
ASSEMBLER | BYTECODE
----------+------------
pushc 2   | 0x01000002
pushc 3   | 0x01000003
mul       | 0x04000000
pushc 5   | 0x01000005
add       | 0x02000000
wrint     | 0x08000000
halt      | 0x00000000
```

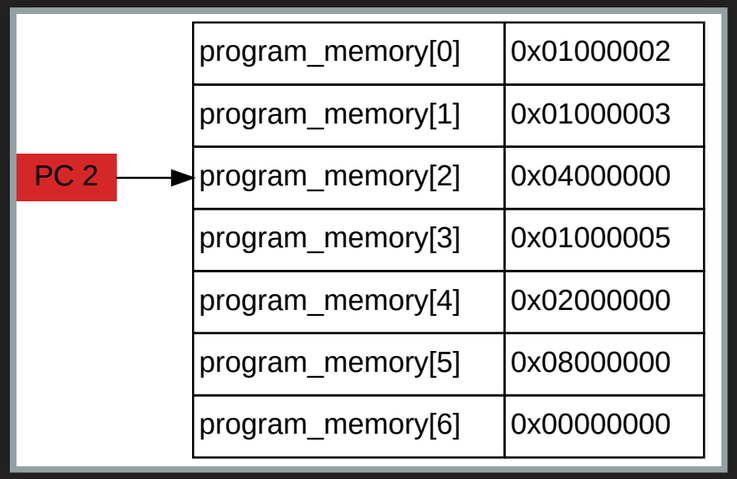
Programmspeicher in C

```c
unsigned int program_memory[] ={
        0x01000002, //program_memory[0]
        0x01000003, //program_memory[1]
        0x04000000, //program_memory[2]
        0x01000005, //program_memory[3]
        0x02000000, //program_memory[4]
        0x08000000, //program_memory[5]
        0x00000000  //program_memory[6]
}
```

PROGRAMMSPEICHER UND PROGRAM COUNTER

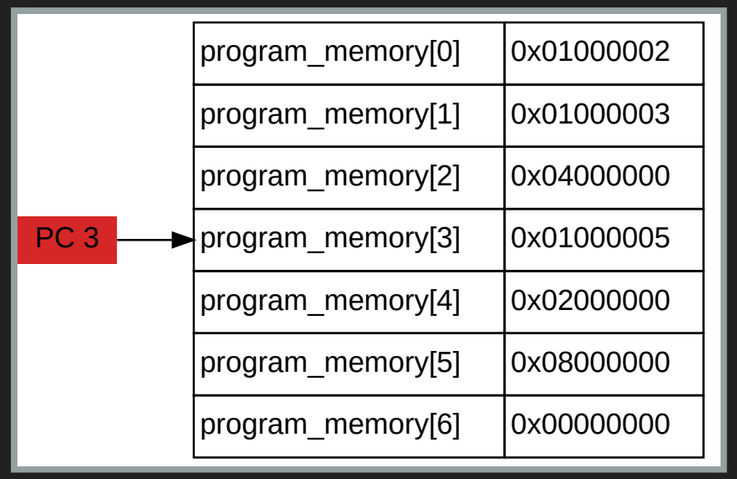
- Um das Programm im Programmspeicher auszuführen, benötigen wir ein zusätzliches Register, das die Stelle im Programm angibt, die ausgeführt werden soll
- Dieses Register ist der sog. Program Counter (PC)

PC auf Position `2` → `mul`



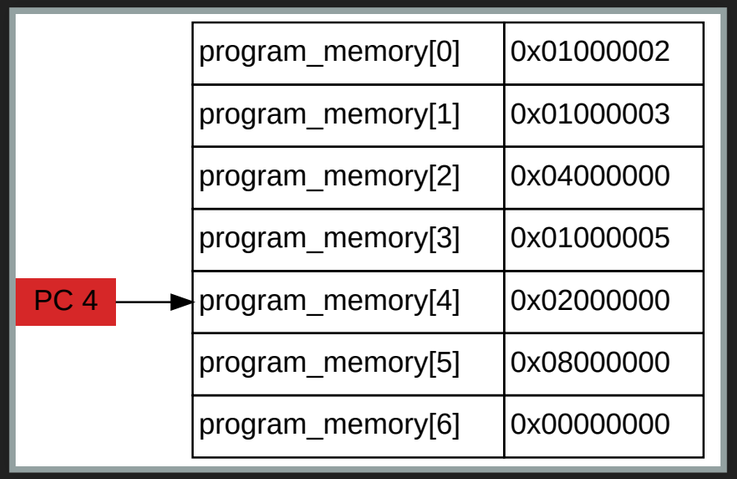

```
ASSEMBLER | BYTECODE
----------+------------
pushc 2   | 0x01000002
pushc 3   | 0x01000003
mul       | 0x04000000  <--
pushc 5   | 0x01000005
add       | 0x02000000
wrint     | 0x08000000
halt      | 0x00000000
```

PC auf Position `2` → `pushc 5`




```
ASSEMBLER | BYTECODE
----------+------------
pushc 2   | 0x01000002
pushc 3   | 0x01000003
mul       | 0x04000000
pushc 5   | 0x01000005 <--
add       | 0x02000000
wrint     | 0x08000000
halt      | 0x00000000
```

PC auf Position `4` → `add`




```
ASSEMBLER | BYTECODE
----------+------------
pushc 2   | 0x01000002
pushc 3   | 0x01000003
mul       | 0x04000000
pushc 5   | 0x01000005
add       | 0x02000000 <--
wrint     | 0x08000000
halt      | 0x00000000
```

PROGRAMMSPEICHER UND PROGRAM COUNTER

- Solange in unseren Programmen keine Sprünge ausgeführt werden, wird der PC nach dem Ausführen einer Instruktion einfach um 1 inkrementiert.
  - Es erfolgt hierbei eine strikt sequentielle Ausführung

Pseudocode Steuerwerk mit PC

```
int pc=0;
while(opcode != HALT) {
        instruction = program_memory[pc];
        pc++;
        execute(instruction);
}
```

Aktuell spielt es **noch** keine Rolle zu welchem Zeitpunkt der PC inkrementiert wird. Sobald Programme mit Sprungbefehlen kommen, wird dies jedoch wichtig! Ein Sprungbefehl ändert den PC, sodass die Ausführung nicht mehr sequentiell abläuft!

- Im Fall, dass das Steuerwerk den PC inkrementiert, wird empfohlen **vor** der eigentlich Ausführung der Instruktion den PC zu inkrementieren
- Das Inkrementieren kann auch Teil der Instruktionsausführung sein. Entscheiden Sie selbst wie Sie es implementieren möchten! Beide Varianten haben vor und Nachteile

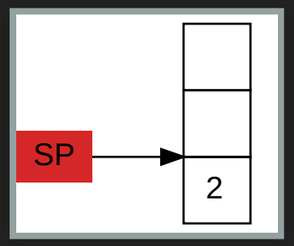
VOLLSTÄNDIGES BEISPIEL: STACK, PM, PC

PC 0

program_memory[0]	0x01000002
program_memory[1]	0x01000003
program_memory[2]	0x04000000
program_memory[3]	0x01000005
program_memory[4]	0x02000000
program_memory[5]	0x08000000
program_memory[6]	0x00000000

```
ASSEMBLER | BYTECODE
----------+------------
pushc 2   | 0x01000002 <--
pushc 3   | 0x01000003
mul       | 0x04000000
pushc 5   | 0x01000005
add       | 0x02000000
wrint     | 0x08000000
halt      | 0x00000000
```

Stack

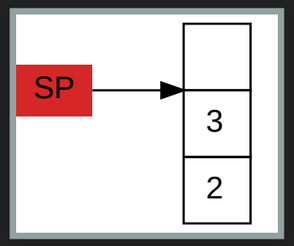


PC 1

program_memory[0]	0x01000002
program_memory[1]	0x01000003
program_memory[2]	0x04000000
program_memory[3]	0x01000005
program_memory[4]	0x02000000
program_memory[5]	0x08000000
program_memory[6]	0x00000000

```
ASSEMBLER | BYTECODE
----------+------------
pushc 2   | 0x01000002
pushc 3   | 0x01000003 <--
mul       | 0x04000000
pushc 5   | 0x01000005
add       | 0x02000000
wrint     | 0x08000000
halt      | 0x00000000
```

Stack

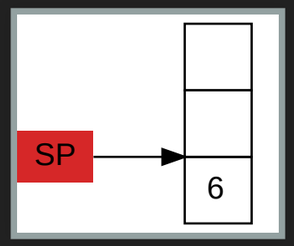


PC 2

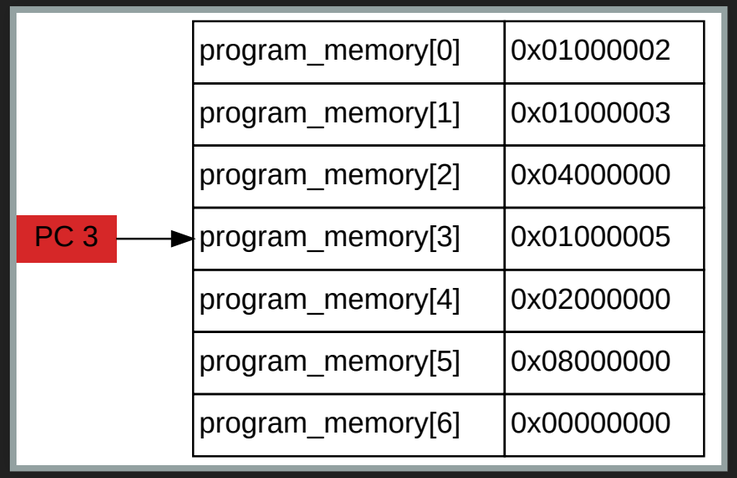
program_memory[0]	0x01000002
program_memory[1]	0x01000003
program_memory[2]	0x04000000
program_memory[3]	0x01000005
program_memory[4]	0x02000000
program_memory[5]	0x08000000
program_memory[6]	0x00000000

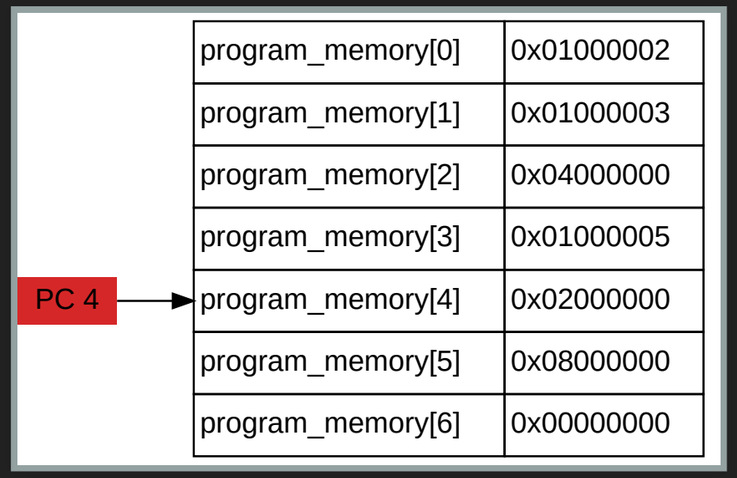
```
ASSEMBLER | BYTECODE
----------+------------
pushc 2   | 0x01000002
pushc 3   | 0x01000003
mul       | 0x04000000 <--
pushc 5   | 0x01000005
add       | 0x02000000
wrint     | 0x08000000
halt      | 0x00000000
```

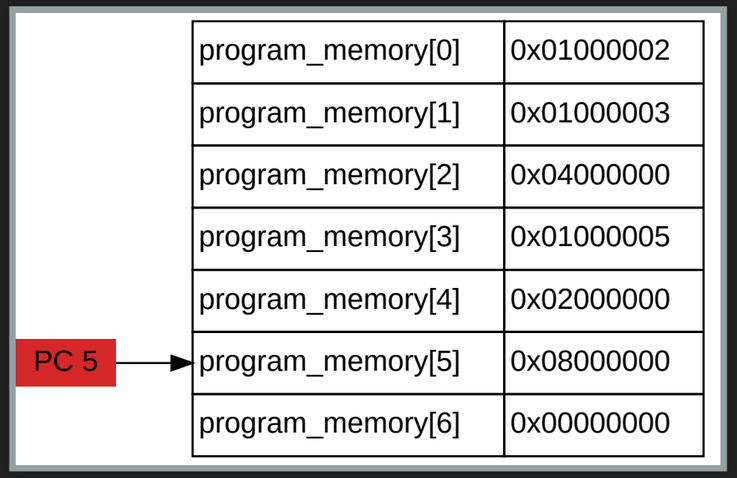
Stack



VOLLSTÄNDIGES BEISPIEL: STACK, PM, PC





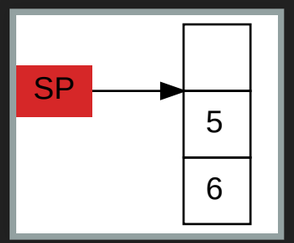



```
ASSEMBLER | BYTECODE
----------+------------
pushc 2   | 0x01000002
pushc 3   | 0x01000003
mul       | 0x04000000
pushc 5   | 0x01000005 <--
add       | 0x02000000
wrint     | 0x08000000
halt      | 0x00000000
```

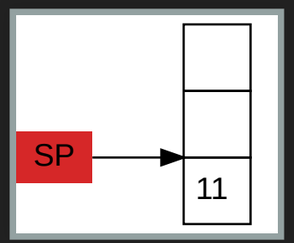
```
ASSEMBLER | BYTECODE
----------+------------
pushc 2   | 0x01000002
pushc 3   | 0x01000003
mul       | 0x04000000
pushc 5   | 0x01000005
add       | 0x02000000 <--
wrint     | 0x08000000
halt      | 0x00000000
```

```
ASSEMBLER | BYTECODE
----------+------------
pushc 2   | 0x01000002
pushc 3   | 0x01000003
mul       | 0x04000000
pushc 5   | 0x01000005
add       | 0x02000000
wrint     | 0x08000000 <--
halt      | 0x00000000
```

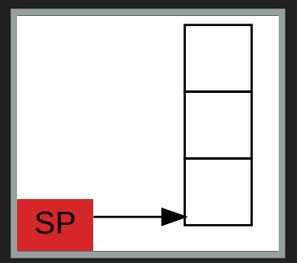
Stack



Stack



Stack



ANMERKUNGEN ZUR AUFGABE 1

Sie haben nun das Handwerkszeug um eine Ninja VM zu programmieren, die eine Ausführung von Programmen in Bytecode erlaubt.

- Implementieren Sie alle Operationen, den Stack und die Register SP und PC
- Verwenden Sie die hier vorgestellten Makros und Verfahren
- Übersetzen Sie die angegebenen Programme per Hand in Bytecode und laden diese in unterschiedliche Programmspeicher
- Ermöglichen Sie die Auswahl einzelner Programme
- Implementieren Sie das Steuerwerk und führen Sie die Programme aus

Testen Sie Ihre VM bereits ausgiebig! Testen Sie insbesondere die Eingabe und die Verarbeitung von negativen Werten (z.B. bei Programm 2). Überlegen Sie sich bereits jetzt eine Strategie, wie Sie Ihr Programm strukturiert und automatisiert testen können. **TIP:** Verwenden Sie zum Testen die Referenzimplementierung und vergleichen die Ausgaben Ihrer VM und der Referenzimplementierung miteinander. Unter Linux hilft Ihnen hierbei die Ausgabeumleitung `./njvm > test.out` und das Programm `diff` hilft Ihnen die Ausgaben zu prüfen. Lesen Sie hierzu die manpage `man diff`.
https://www.geeksforgeeks.org/diff-command-linux-examples/
https://www.geeksforgeeks.org/input-output-redirection-in-linux/