1. Universalrechner, von Neumann vs. Harvard

a) Der wesentliche Unterschied ist, dass sich bei von Neumann Programmcode und Daten denselben Speicherbereich teilen, während im Harvard-Modell getrennte Bereiche genutzt werden, die über separate Busse an die CPU angebunden sind.

Vorteile von Neumann: einfacher umzusetzen; selbstmodifizierende Programme möglich

Nachteile von Neumann: Kein gleichzeitiger Zugriff auf Programmcode und Daten (da im selben Speicher), dadurch Pipelining schwieriger

b) Ob die Trennung zwischen CPUs und GPUs noch sinnvoll ist, kann man aus verschiedenen Perspektiven sehen.
1) Kann eine GPU eine CPU vollständig ersetzen? (eher nein, aber vielleicht bald),
2) Für welche Zwecke werden CPUs/GPUs hergestellt? Hieraus folgt auch, für welche Aufgaben diese Prozessoren optimiert sind.

2. Spezialregister

a) Mit JUMP-Befehlen ändern Sie direkt den Wert des PC.

b) Der Stack-Pointer zeigt auf den obersten Eintrag im Stack. Wenn Sie den Wert willkürlich verändern, passen z. B. Parameter und Rücksprungadressen des aktuellen Funktionsaufrufs nicht mehr zum „neuen Stack“ bzw. Sie verlieren den Zugriff auf mit PUSH abgelegte Daten.

Die Befehle PUSH und POP beeinflussen allerdings auch direkt den SP, aber immer nur in Form von kleinen Additionen/Subtraktionen (entsprechend der Größe der auf dem Stack abgelegten oder heruntergenommenen Daten).

3. Stack-Maschine

```
PUSH 0x1000
PUSH 0x1000
MUL             (jetzt: x^2)
PUSH 0x1000
MUL             (jetzt: x^3)
PUSH 0x1000
SUB
PUSH 0x1000
SUB             (jetzt: x^3-2x)
PUSH 0x1001
ADD             (jetzt: x^3-2x+a)
PUSH 0x1000
PUSH 0x1002
SUB             (oben auf dem Stack: x-b, darunter: x^3-2x+a)
DIV             Endergebnis
```

4. Register-Memory-Befehle

Alles in „Pseudo-MMIX-Syntax“:

a) ADD $0, (Adresse) wird:

```
LDB $201, Adresse
ADD $0,$0,$201
```

b) ADD (Adr1, Adr2) wird:

```
LDB $201, Adr1
LDB $202, Adr2
ADD $201,$201,$202
STB Adr1, $201
```

c) JPEQ (Adr1), (Adr2), ZielAdr wird:

```
LDB $201, Adr1
LDB $202, Adr2
SUB $201,$201,$202
BZ $201, ZielAdr
```

5. Register-Memory-Befehle

a) Zuerst „ohne Optimierung“:

```
LOAD 0x1000       (x)
MUL 0x1000        (x^2)
MUL 0x1000        (x^3)
SUB 0x1000        (x^3-x)
SUB 0x1000        (x^3-2x)
ADD 0x1001        (x^3-2x+a)
STOR 0x1004       (zwischenspeichern)
LOAD 0x1000       (x)
SUB 0x1002        (x-b)
STOR 0x1005       (zwischenspeichern)
LOAD 0x1004       (x^3-2x+a)
DIV 0x1005        (x^3-2x+a) / (x-b)
```

Effizienter: Erst den Nenner ausrechnen und zwischenspeichern, dann den Zähler berechnen und durch den Nenner teilen (spart eine Speicherzelle und ein STOR/LOAD-Paar):

```
LOAD 0x1000       (x)
SUB 0x1002        (x-b)
STOR 0x1005       (zwischenspeichern)
LOAD 0x1000       (x)
MUL 0x1000        (x^2)
MUL 0x1000        (x^3)
SUB 0x1000        (x^3-x)
SUB 0x1000        (x^3-2x)
ADD 0x1001        (x^3-2x+a)
DIV 0x1005        (x^3-2x+a) / (x-b)
```

b) Rechnen mit Registern ist schneller als Zugriff auf Speicher.

c) Zwei Interpretationen der Aufgabe: Akku-Verhalten in MMIX nachbilden oder richtiges MMIX-Programm...

In „Pseudo-MMIX-Syntax“:

```
x IS $1
a IS $2
b IS $3
MUL $4,x,x        (x^2)
MUL $4,$4,x       (x^3)
SUB $4,$4,x       (x^3-x)
SUB $4,$4,x       (x^3-2x)
ADD $4,$4,a       (x^3-2x+a)
SUB $5,x,b        (x-b)
DIV $4,$4,$5      (Ergebnis: Bruch)
```