# Technische Informatik I
# Mikrocode-basierter CPU-Simulator

Thorsten Thormählen

26. Januar 2021

Teil 9, Kapitel 1

Dies ist die Druck-Ansicht.

Aktiviere Präsentationsansicht

## Steuerungstasten

→ nächste Folie (auch Enter oder Spacebar).
← vorherige Folie
d schaltet das Zeichnen auf Folien ein/aus
p wechselt zwischen Druck- und Präsentationsansicht
CTRL + vergrößert die Folien
CTRL - verkleinert die Folien
CTRL 0 setzt die Größenänderung zurück

# Notation

| Typ | Schriftart | Beispiele |
|---|---|---|
| Variablen (Skalare) | kursiv | $a, b, x, y$ |
| Funktionen | aufrecht | $\mathrm{f}, \mathrm{g}(x), \max(x)$ |
| Vektoren | fett, Elemente zeilenweise | $\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top$, $\mathbf{B} = (x, y, z)^\top$ |
| Matrizen | Schreibmaschine | $\mathtt{A}, \mathtt{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ |
| Mengen | kalligrafisch | $\mathcal{A}, B = \{a, b\}, b \in \mathcal{B}$ |
| Zahlenbereiche, Koordinatenräume | doppelt gestrichen | $\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$ |

# CPU-Simulator

In diesem Kapitel wird anhand eines vereinfachten Modells die Funktionsweise einer CPU (Central Processing Unit) erläutert

Das Modell hat nur geringe Ähnlichkeit mit existierenden Prozessoren

Trotzdem ist es voll funktionsfähig

Dieses Modell-CPU entstand im Rahmen von Vorlesungen, die Peter Gumm und Manfred Sommer hier in Marburg gehalten haben

Von Martin Perner wurde ein entsprechendes Simulationsprogramm namens MikroSim erstellt, das lange in dieser Vorlesung eingesetzt wurde: MikroSim

Das Modell und Programm wurden 1995 auf der CeBit vorgestellt:

H. Peter Gumm, Martin Perner: *Der Mikrocodesimulator MicroSim*. CeBit 1995

Seit WS 2013/14 werden in der Vorlesung und der Übung eine alternative Implementierung in Form einer Web-Applikation eingesetzt: CPU-Simulator

# CPU-Simulator

Die wichtigsten Einzelteile, aus denen eine CPU aufgebaut ist, wurden bereits besprochen:

- ALU (Arithmetic Logic Unit)
- Register
- Speicher

Diese Komponenten sind durch Leitungen verbunden, die durch steuerbare Schalter geöffnet oder geschlossen werden können.

Das Öffnen und Schließen dieser Schalter muss in einer zeitlichen Abfolge koordiniert werden

Daher besitzt eine CPU zunächst einen Taktgeber, der die Zeit in einzelne Takte unterteilt

Diese Takte sind sehr kurz, bei einem 1-GHz-Prozessor dauert ein Takt $10^{-9}$s, also eine Nanosekunde

In den folgenden Folien wird eine sehr einfache, idealisierte Definition eines CPU-Taktes beschrieben

Was genau in einem Takt passiert kann bei realen Prozessoren sehr unterschiedlich sein

## Takt und Takt-Phasen

Jede Operation der CPU benötigt einen Takt

Für eine einfache Operation, wie etwa die Addition zweier Registerinhalte, werden dazu drei Phasen benötigt:

Phase 1: Hol-Phase (engl. "fetch")
Holt die Argumente aus den Registern und stelle sie der ALU bereit

Phase 2: Rechenphase (engl. "execute")
Führt die ALU-Operation durch

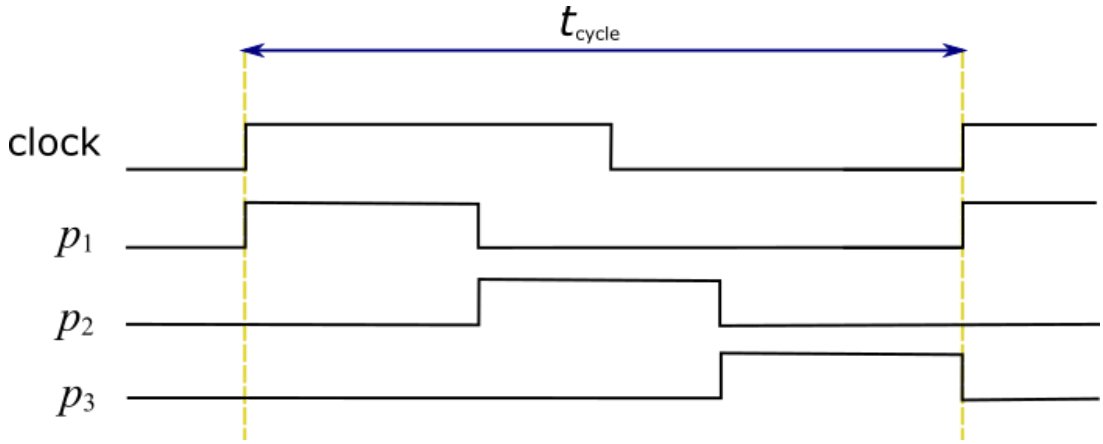
Phase 3: Bring-Phase (engl. "store")
Speichert das Ergebnis in ein Register

# Takt-Phasen

Bei unserer Modell-CPU müssen für jede dieser Phasen gewisse Schalter geöffnet, andere wieder geschlossen werden

Daher arbeitet die CPU intern mit drei Phasen-Steuersignalen $p_1$, $p_2$, $p_3$, die abwechselnd auf 1, dann wieder auf 0 gesetzt werden

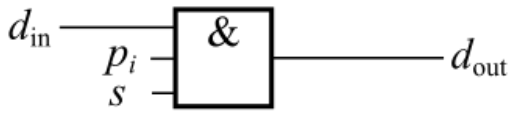
In Phase $i$ ist $p_i = 1$, alle anderen $p_{j \neq i} = 0$



# Steuerbare Schalter

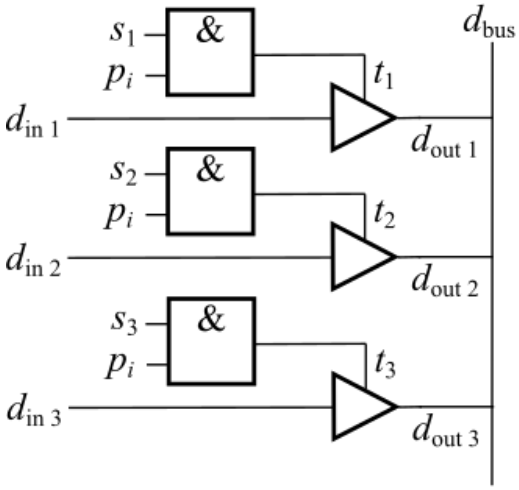
Damit die Datenleitungen zur richtigen Zeit offen bzw. geschlossen sind, werden sie durch Schalter gesichert, die nur für eine bestimmte Phase geöffnet werden können

Ein solcher Schalter könnte z.B. durch ein UND-Gatter mit 3 Eingängen realisiert werden




Dieser Schalter würde nur bei gesetztem Steuersignal $s = 1$ und nur in Phase $i$ (d.h. $p_i = 1$) die Daten vom Eingang $d_{\text{in}}$ an den Ausgang $d_{\text{out}}$ weiterleiten

In der Praxis werden zum Ansteuern von Datenleitungen auch gerne so genannte Tristate Buffer eingesetzt, die eine Datenleitung bei $t = 0$ hochohmig schalten. D.h. es gibt 3 Zustände am Ausgang: 0, 1 und hochohmig.



# Mikrobefehl

In unserer beispielhaften Modell-CPU finden sich zahlreiche der eben beschriebenen Schalter

Diese bleiben für einen Takt lang in einer bestimmten Einstellung und sind ggf. in bestimmten Phasen des Taktes offen

Die Stellung eines Schalters wird durch sein Steuersignal $s_j$ definiert

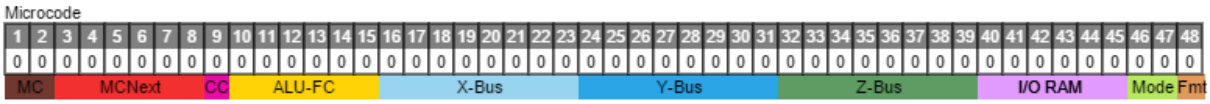
In einem so genannten Mikrobefehlswort oder Mikrobefehl wird die Stellung vieler Schalter bzw. Steuersignale zusammengefasst

Eine Folge von Mikrobefehlen wird auch als Mikrocode bezeichnet

Ein Mikrobefehlswort bleibt einen Takt lang gültig und steuert in diesem Takt die Arbeitsweise der CPU

In unserer Modell-CPU besteht jedes Mikrobefehlswort aus genau 48 Bits:
$s_1, s_2, \ldots, s_{48}$

Die Bedeutung dieser 48 Bits werden auf den folgenden Folien schrittweise erläutert



# Register und Busse

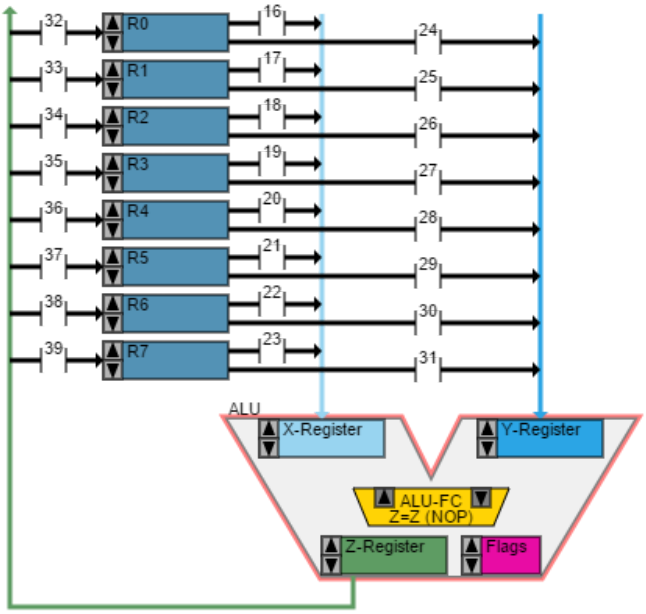
Register enthalten Datenworte, d.h. aus mehreren Bits bestehende Daten

In unserer Modell-CPU werden Register als farbige Rechtecke dargestellt

Wir beschreiben eine 16-Bit-Architektur, so dass alle Register 16 Bit breit sind

Der Registerinhalt wird jeweils über eine 4-stellige Hexadezimal-Zahl dargestellt

Datenleitungen verbinden Register miteinander




Bei einem Registertransfer werden die entsprechenden Bits von Quell- und Zielregister durch parallele Leitungen (Datenbus) verbunden

In unserer Modell-CPU wird der Datenbus zwischen zwei Registern als eine etwas dickere Linie mit einem Pfeil dargestellt, der die Richtung des Datentransfers anzeigt

# Bus-Schalter

Bus-Schalter schalten alle Datenleitungen für ein Register gemeinsam (16 parallele Schalter mit einem gemeinsamen Steuersignal)

Ein geöffneter Bus-Schalter wird durch zwei dünnere parallel Linien dargestellt

Bei einem geschlossenem Bus-Schalter sind die dünneren parallelen Linien mit einem Bus verbunden

Die Bus-Schalter bleiben für einen Takt lang in einer bestimmten Einstellung und sind ggf. in bestimmten Phasen des Taktes offen

Die Stellung eines Schalters wird durch sein Steuersignal $s_j$ definiert, dabei ist der Index $j$ jeweils als Zahl leicht oberhalb des Schalters zu finden

- Die kleine Abbildung rechts zeigt den Bus-Schalter, der dem Steuerbit $s_{32}$ des Mikrobefehlsworts gehorcht
- Links geöffnet $s_{32} = 0$, rechts geschlossen $s_{32} = 1$


Selbst ausprobieren:

- Öffnen des CPU-Simulators
- Klicken auf den Schalter 32 ändert dessen Schalterstellung und Steuerbit $s_{32}$
- Klicken auf ein Steuerbit ändert ebenfalls die Schalterstellung

# CPU-Simulator: Komplexitätsstufe 1

# CPU-Simulator: Komplexitätsstufe 1

Nun wurden alle Bauelemente beschrieben, um einen Taschenrechner mit einigen Speicherzellen (Registern) zu bauen

Wir benötigen dazu zunächst eine ALU, eine Reihe von Registern (hier $R_0$, $R_1$, ..., $R_7$),

Busse und Bus-Schalter

Die ALU versehen wir mit zwei Operandenregistern, X und Y, sowie einem Ergebnisregister Z

Dann verbinden wir jedes Allzweckregister $R_0$, $R_1$, ..., $R_7$ über zwei Busse (dem X-Bus

und dem Y-Bus) mit den entsprechenden Operandenregistern der ALU

Das Ergebnisregister Z der ALU wird über den Z-Bus mit den Allzweckregistern verbunden

# CPU-Simulator: Komplexitätsstufe 1

Zwischen den Registern und den Bussen sitzen Schalter, die nur in bestimmten Phasen geöffnet werden können (`fetch`, `execute`, `store`).

Die Schalter zwischen den Allzweckregistern und dem X- und Y-Bus sind nur in der Hol-Phase `fetch` aktiv

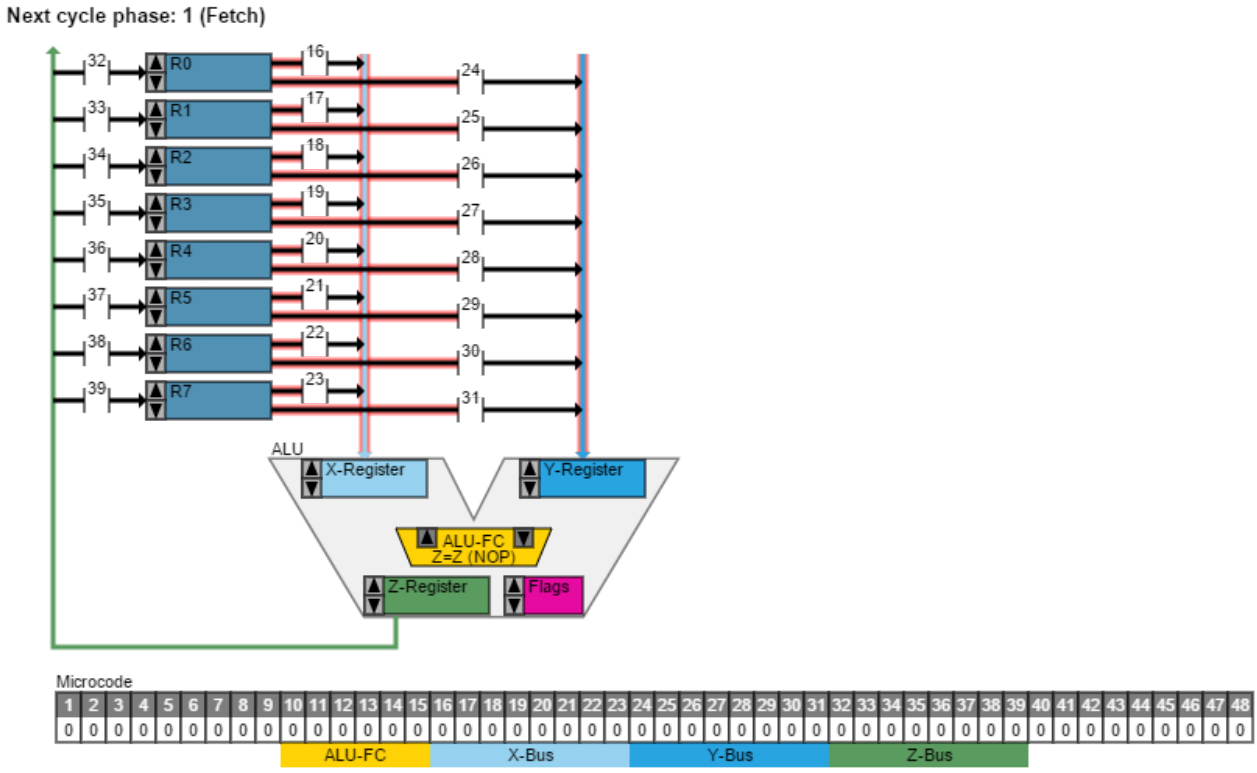
Nur dann können die Daten von den Registern zu den Operandenregistern der ALU fließen

In der zweiten Phase `execute` rechnet die ALU und schreibt das Ergebnis in `Z`

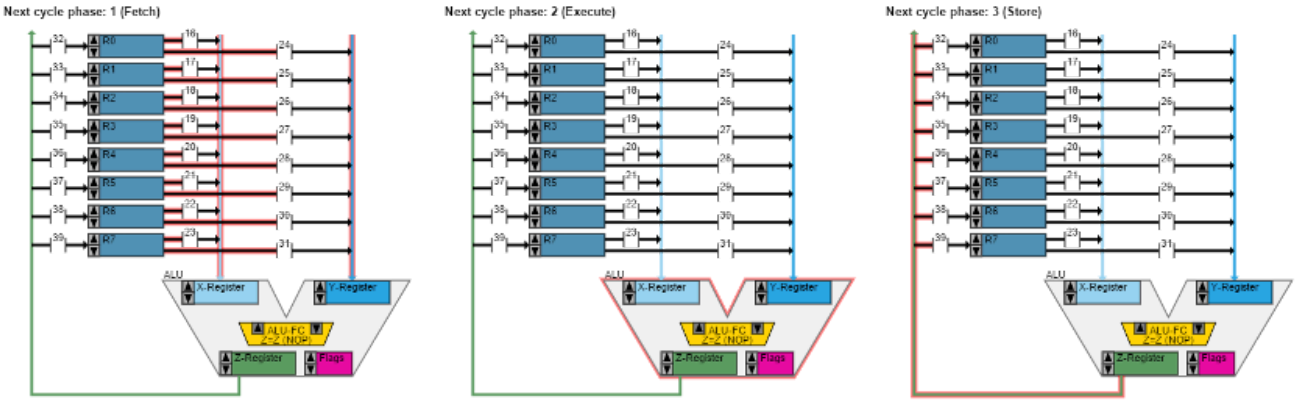
Nur in dieser Bring-Phase `store` sind die Schalter zwischen Z-Bus und den Allzweckregistern aktiv, damit das Ergebnis in einem der Register abgelegt werden kann

Der komplette Aufbau ist in der Abbildung auf der nächsten Folie dargestellt

# CPU-Simulator: Komplexitätsstufe 1



# CPU-Simulator: Komplexitätsstufe 1




Um zu verdeutlichen, in welcher Phase (`fetch`, `execute`, `store`) welche Komponenten aktiv sind, werden diese bei Aktivität jeweils rot unterlegt

Selbst ausprobieren:

Öffnen des CPU-Simulators

Klicken auf den Druckknopf "Next" aktiviert die nächste Phase

# Mikrocodegesteuerte Operationen

Schließlich muss noch an der ALU einstellt werden, welche Operation sie berechnen soll

Wir nehmen an, dass unsere ALU ein Repertoire von 64 Operationen umfasst, so dass wir die Operation mit 6 Bit im Mikrobefehlswort einstellen können

Diese 6 Bit steuern einen Multiplexer (dargestellt als symmetrisches Trapez) in der ALU, der die Operation auswählt

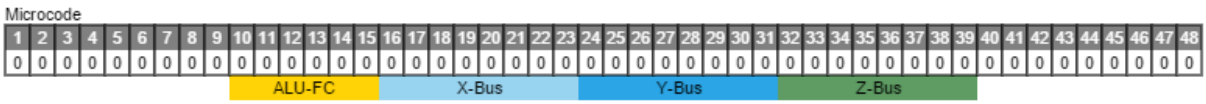
Neben den grundlegenden arithmetischen, logischen und vergleichenden Operationen sind auch Operationen mit Konstanten möglich

Diese dienen z.B. dazu, eine feste Konstante im Z-Register bereitzustellen

Die ALU-Funktionscodes (`ALU-FC`) sind in den folgenden Tabellen dargestellt

In der Operations-Spalte wird angegeben, wie sich der Wert im Z-Register aus den Werten in den X- und Y-Registern ergibt

Ggf. werden noch die Inhalte von X- und Y-Register einander zugewiesen bzw. vertauscht, was durch $Y \rightarrow X$ bzw. durch $X \leftrightarrow Y$ angedeutet wird



# ALU Operationen

| ALU-FC (Dezimal) | ALU-FC (Binär) | Operation |
|---|---|---|
| 0 | 000000 | $Z = Z$ (Keine Operation) |
| 1 | 000001 | $Z = -Z$ |
| 2 | 000010 | $Z = X$ |
| 3 | 000011 | $Z = -X$ |
| 4 | 000100 | $Z = Y$ |
| 5 | 000101 | $Z = -Y$ |
| 6 | 000110 | $Z = Y, X \leftrightarrow Y$ |
| 7 | 000111 | $Z = X, X \leftrightarrow Y$ |
| 8 | 001000 | $Z = X, Y \rightarrow X$ |
| 9 | 001001 | $Z = X + 1$ |
| 10 | 001010 | $Z = X - 1$ |
| 11 | 001011 | $Z = X + Y$ |
| 12 | 001100 | $Z = X - Y$ |
| 13 | 001101 | $Z = X * Y$ |
| 14 | 001110 | $Z = X \operatorname{div} Y$ |
| 15 | 001111 | $Z = X \bmod Y$ |

# ALU Operationen

| ALU-FC (Dezimal) | ALU-FC (Binär) | Operation |
|---|---|---|
| 16 | 010000 | $Z = X \operatorname{sal} Y$ (shift arithmetic left ) |
| 17 | 010001 | $Z = X \operatorname{sar} Y$ (shift arithmetic right ) |
| 18 | 010010 | $Z = X \operatorname{cmpa} Y$ (compare arithmetic) |
| 19 | 010011 | $Z = X \operatorname{and} Y$ |
| 20 | 010100 | $Z = X \operatorname{nand} Y$ |
| 21 | 010101 | $Z = X \operatorname{or} Y$ |
| 22 | 010110 | $Z = X \operatorname{nor} Y$ |
| 23 | 010111 | $Z = X \operatorname{xor} Y$ |
| 24 | 011000 | $Z = X \operatorname{nxor} Y$ |
| 25 | 011001 | $Z = X \operatorname{sll} Y$ (shift logic left) |
| 26 | 011010 | $Z = X \operatorname{slr} Y$ (shift logic right) |
| 27 | 011011 | $Z = X \operatorname{cmpl} Y$ (compare logic) |
| 28 | 011100 | $X = 0$ |
| 29 | 011101 | $X = (\mathrm{FFFF})_{16}$ |
| 30 | 011110 | $Y = 0$ |
| 31 | 011111 | $Y = (\mathrm{FFFF})_{16}$ |

# ALU Operationen

| ALU-FC | Operation |
|---|---|
| Falls 32 ≤ ALU-FC < 48 | Z = X = ALU-FC - 32 |
| Falls 48 ≤ ALU-FC < 64 | Z = Y = ALU-FC - 48 |

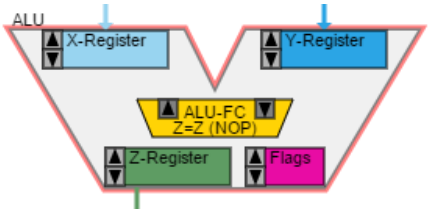
Beispiele:

Für ALU-FC = 35 ergibt sich Z = X = 3

Für ALU-FC = 63 ergibt sich Z = Y = 15

# Statusregister (Flags)

Führt die ALU eine Berechnung durch, wird nicht nur das Ergebnis im Z-Register abgelegt, sondern auch das Flag-Register gesetzt




Die ALU signalisiert damit, dass bestimmte Ereignisse bei der Berechnung aufgetreten sind

In unserer Modell-CPU sind dies 4 verschiedene, daher ist das Flag-Register 4 Bit breit:

- Bit 0: Overflow einer arithmetischen Operation (overflow flag)
- Bit 1: Ergebnis war negativ (sign neg flag)
- Bit 2: Ergebnis war positiv (sign pos flag)
- Bit 3: Ergebnis war null (zero flag)

Der Inhalt des Flag-Registers wird als Hexadezimal-Zahl dargestellt

Beispiele:

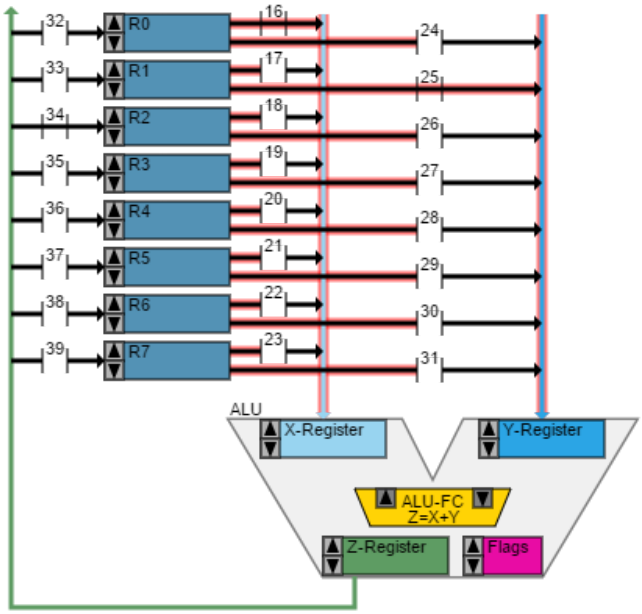
- Flag = $(5)_{16} = (0101)_2$, d.h. ein positives Ergebnis und Overflow sind aufgetreten
- Flag = $(8)_{16} = (1000)_2$, d.h. das Ergebnis ist null

# Beispiel: Addition zweier Zahlen

Als Beispiel für die Verwendung der Modell-CPU soll die Addition zweier Register und die Speicherung des Ergebnisses betrachtet werden:
$R_3 = R_1 + R_2$

Dazu muss ALU-FC Nr. 11 = $(001011)_2$ eingestellt sein

Die Steuersignale für den X-Bus sind 10000000, für den Y-Bus 01000000 und für den Z-Bus 00100000




Microcode

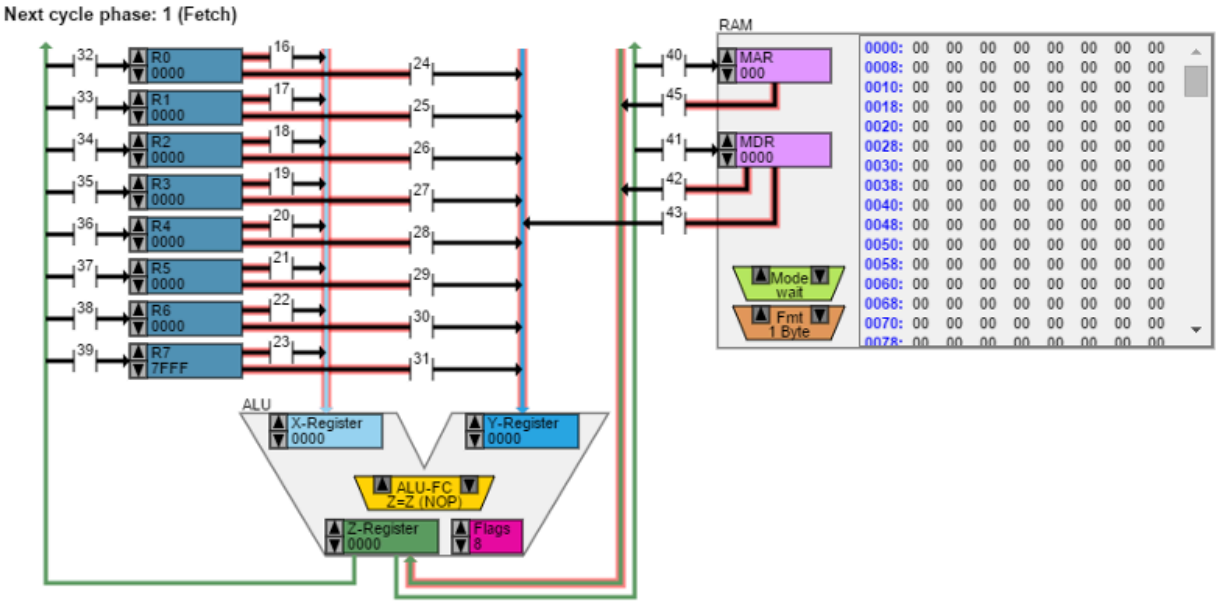
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

ALU-FC | X-Bus | Y-Bus | Z-Bus

# CPU-Simulator: Komplexitätsstufe 2

# CPU-Simulator: Komplexitätsstufe 2

Die CPU wird nun um einen Hautspeicher (RAM) erweitert



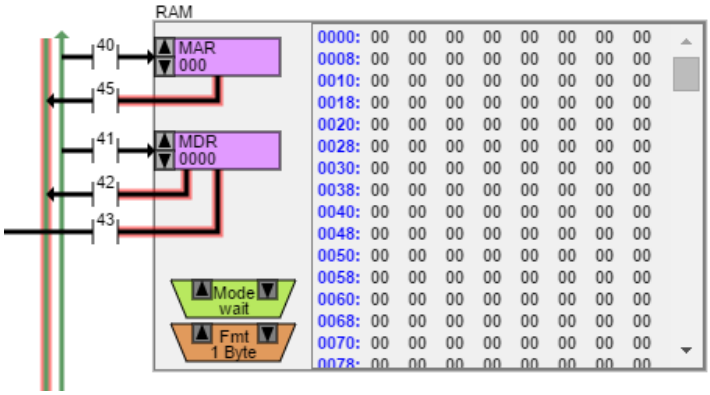
# Hauptspeicher

Der Hauptspeicher (engl. Random Access Memory kurz RAM) hat als Schnittstelle zwei Register, eine Formatangabe und einen einstellbaren Modus

Bei den Registern handelt es sich um das Adressregister (`MAR` = Memory Address Register) und das Datenregister (`MDR` = Memory Data Register)

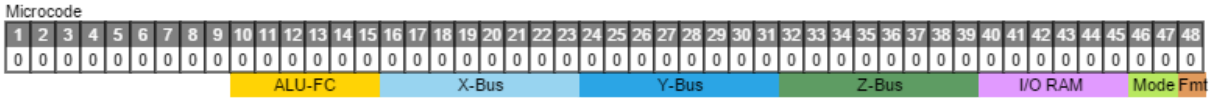
Wie bereits im Kapitel über Speicher besprochen, steht im Adressregister eine Speicheradresse und im Datenregister ein Wert, der an der angegebenen Adresse geschrieben werden soll oder von der angegebenen Adresse gelesen wurde

Insgesamt umfasst der RAM Speicher 1024 Bytes, $(000)_{16}$ bis $(3FF)_{16}$



# Mode und Format

Um das RAM per Mikrocode anzusteuern, wird das Mikrobefehlswort erweitert




Die Bits 40 bis 45 (`I/O RAM`) steuern die Bus-Schalter der RAM-Register

Die Bits 46 und 47 (`Mode`) steuern den I/O Mode:

- 00 = wartend
- 01 = lesend
- 10 = schreibend

Und Bit 48 (`Fmt`), wählt das Datenformat (8 oder 16 Bit Lesen/Schreiben)

- 0 = 1 Byte Lesen/Schreiben
- 1 = 2 Byte Lesen/Schreiben

Beim Schreiben einer 8-Bit-Größe wird nur das niederwertige Byte aus dem `MDR` an die Adresse `[MAR]` geschrieben.

Beim Lesen werden die vorderen 8 Stellen des `MDR` durch Nullen aufgefüllt

Beim 16 Bit Lesen/Schreiben wird das Big-Endian-Format verwendet

# Schreiben in den Hauptspeicher

Es werden in der Regel zwei Takte benötigt

Die Adresse im Hauptspeicher, an die geschrieben werden soll, wird meist mit Hilfe der ALU berechnet

Im ersten Takt muss die Adresse, an die geschrieben werden soll, in Phase 3 über den Z-Bus in das Adressregister `MAR` transportiert werden

Im zweiten Takt muss der Datenwert, der in den Speicher geschrieben werden soll, in Phase 3 über den Z-Bus in das Datenregister `MDR` transportiert werden

Nach Takt 2, Phase 3 hat der Hauptspeicher die Schreiboperation durchgeführt

Selbst ausprobieren:

- Öffnen des CPU-Simulators
- In der Combo-Box "Import example file" das Beispiel "Write to RAM" auswählen
- Das Beispiel schreibt $(000F)_{16}$ an die Speicherstelle $(008)_{16}$

# Lesen vom Hauptspeicher

Es werden in der Regel zwei Takte benötigt

Im ersten Takt muss die Adresse, von der gelesen werden soll, in Phase 3 über den Z-Bus in das Adressregister MAR transportiert werden

Im zweiten Takt, Phase 1 steht der gelesene Wert im Datenregister MDR zur Verfügung

Die Daten des MDR können wahlweise über den Y- bzw. über den Z-Bus in das Y- und/oder in das Z-Register übertragen werden

Vom Z-Register aus können sie in Takt 2, Phase 3 in eines der Register geschrieben werden

Vom Y-Register aus können sie in Takt 2, Phase 2 in der ALU weiterverarbeitet werden

Selbst ausprobieren:

- Öffnen des CPU-Simulators
- In der Combo-Box "Import example file" das Beispiel "Read from RAM" auswählen
- Das Beispiel liest $(0102)_{16}$ von der Speicherstelle $(010)_{16}$

# Idealisierter Hauptspeicher

Der in unserer Modell-CPU verwendete Hauptspeicher liest und schreibt ohne Verzögerung

Die ist eine idealisierte Annahme

In der Realität kommt es häufig vor, dass der Speicher für die CPU zu langsam ist

Die CPU muss ggf. ein oder mehrere Takte warten (waitstates), um dem Speicher Zeit zu geben, die Daten zu lesen bzw. zu schreiben

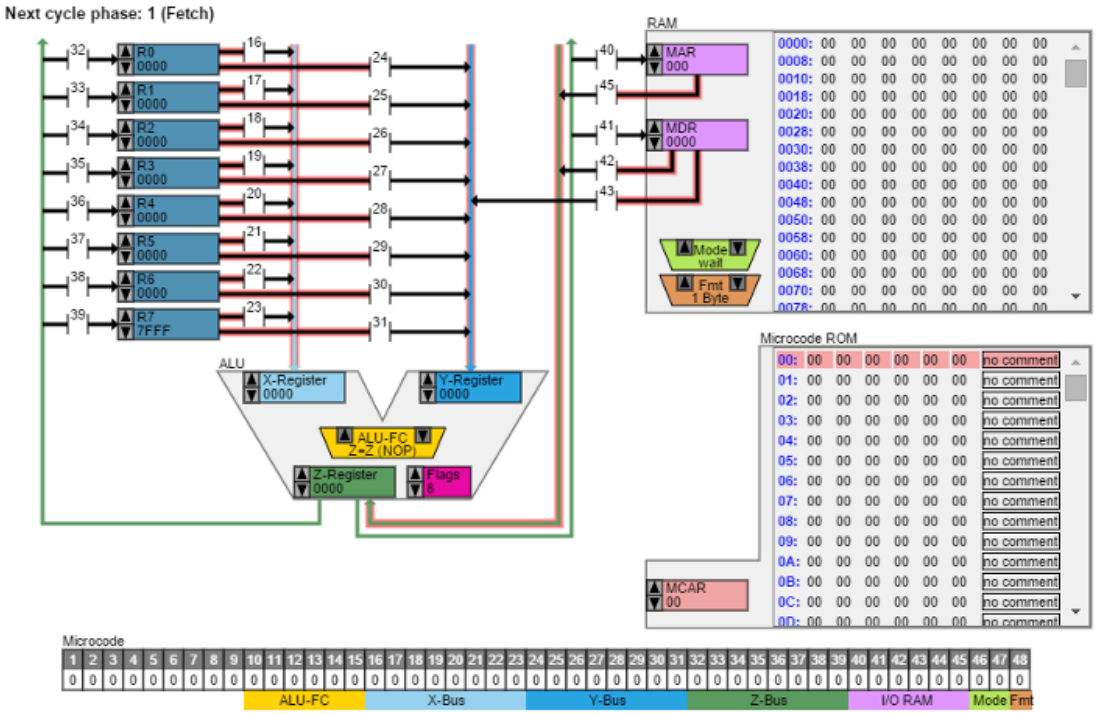
Die Anzahl der benötigten Warte-Takte kann sehr unterschiedlich sein, je nachdem ob die Daten aus einem schnellen oder langsamen Cache kommen oder womöglich sogar aus dem noch langsameren Hauptspeicher

Während der Warte-Takte kann die CPU jedoch (falls möglich) andere unabhängige Berechnungen ausführen

# CPU-Simulator: Komplexitätsstufe 3

# CPU-Simulator: Komplexitätsstufe 3

Die CPU wird nun um einen Mikrobefehlsspeicher erweitert



# Der Mikrobefehlsspeicher

Mikrobefehle sind Bitfolgen, die wie andere Daten auch in einem Speicher abgelegt werden können

Ein solcher Mikrobefehlsspeicher ist Teil der CPU

Der Speicher ist als ROM (Read-Only-Memory) ausgeführt, d.h. er kann nur gelesen, nicht aber von der CPU nicht verändert werden

Ansonsten ist das ROM wie jeder andere Speicher aufgebaut, insbesondere besitzt es ein Adressregister, in dem die Adresse eines Speicherwertes abgelegt wird, und ein Datenregister, in dem der dort befindliche Datenwert zurückgegeben wird

Weil die im ROM gespeicherten Daten als Mikrocode interpretiert werden, wird das Adressregister mit `MCAR` (MicroCode Address Register) bezeichnet

Das Datenregister ist im Simulator nicht als farbiges Rechteck dargestellt, da dessen Inhalt immer dem aktuellen Mikrocode entspricht, dessen 48 Bits sowieso jeweils unten dargestellt werden

# Der Mikrobefehlsspeicher

Da in unserem CPU-Modell bis zu 256 Mikrobefehle im ROM speichern werden können, wird ein 8 Bit breites `MCAR` benötigt

Da jeder Mikrobefehl 48 Bits lang ist, wird das auf der CPU befindliche ROM eine Größe von 256 x 48 Bit = 1536 Bytes besitzen

Wir können aber nicht jedes Byte adressieren, wie im RAM, sondern nur jeden Mikrobefehl, d.h. jeweils Worte mit 48 Bit

Ändert sich das Mikrocode-Adressregister wird das entsprechende Mikrocodewort gelesen und dessen Bits beeinflussen sofort als Steuersignale die Bus-Schalter und Multiplexer der restlichen CPU

Mit dem ROM kann die CPU nun komplexere Operationen über mehrere Takte ausführen, indem nach jedem Takt ein anderes Mikrocodewort aus dem ROM-Speicher angewendet wird

Zur Zeit muss das Mikrocode-Adressregister dazu noch von Hand erhöht werden

# Der Mikrobefehlsspeicher

Beispiel: [5,6] = R1 + [5,6]

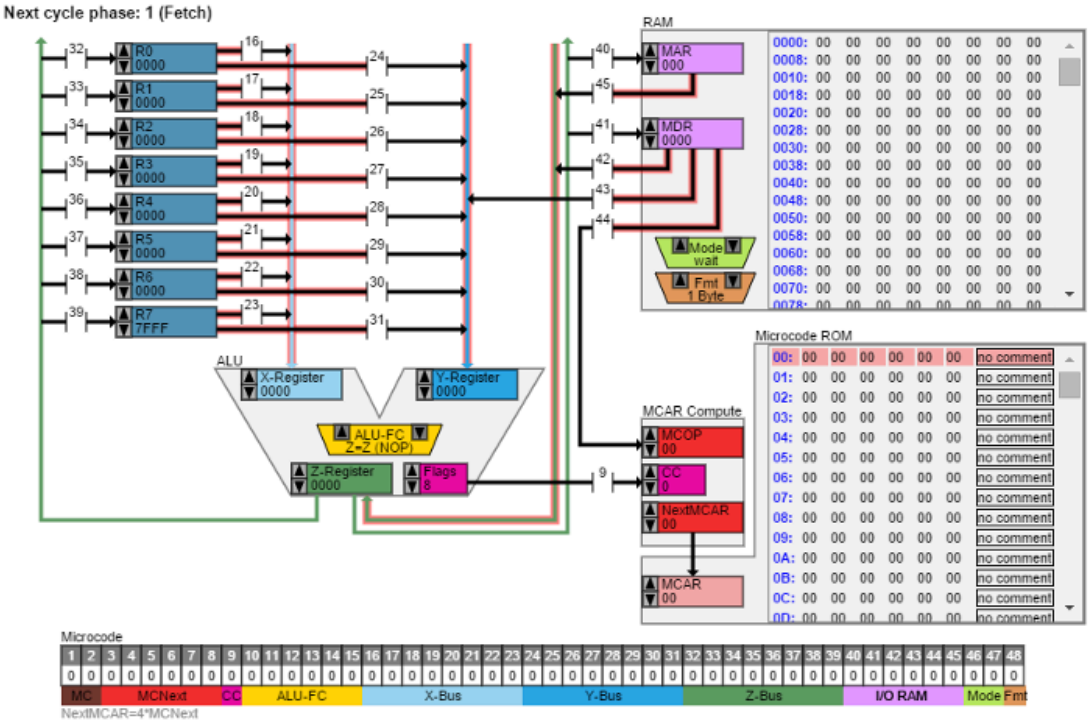
| | | | |
|---|---|---|---|
| Takt 1 | Phase 1 | keine Operation | |
| | Phase 2 | `Z = 5` | `ALU-FC: 100101` |
| | Phase 3 | `MAR = Z` | `I/O RAM: 100000`<br>`Mode: 01, Format: 1` |
| Takt 2 | Phase 1 | RAM ließt Inhalt von [5,6] ins `MDR` | `Mode: 01, Format: 1` |
| | Phase 2 | keine Operation | `ALU-FC: 000000` |
| | Phase 3 | keine Operation | `I/O RAM: 000000` |
| Takt 3 | Phase 1 | `Y = MDR, X = R1` | `X-Bus: 01000000` |
| | Phase 2 | `Z = X + Y` | `ALU-FC: 001011` |
| | Phase 3 | `MDR = Z`, RAM schreibt | `I/O RAM: 010100`<br>`Mode: 10, Format: 1` |

Selbst ausprobieren:
Im CPU-Simulator das Beispiel "Modify RAM: [5] = R1 + [5]" auswählen

# CPU-Simulator: Komplexitätsstufe 4

# CPU-Simulator: Komplexitätsstufe 4

Die CPU wird nun um einen Adressrechner für den Mikrobefehlsspeicher erweitert



# Sprünge im Mikrobefehlsspeicher

Die im ROM befindlichen Befehle könnten von der CPU der Reihe nach abgearbeitet werden

Das wäre aber sehr eintönig und sinnlos, denn dann würde immer dasselbe Programm ablaufen, da der Inhalt des ROM ja unveränderbar ist

Daher ist die Modell-CPU so konstruiert, dass wir jeweils beliebige Mikrocode-Adressen in das `MCAR` schreiben können, um somit beliebige Sprünge im Mikrocode zu realisieren

Während die CPU den gegenwärtigen Mikrobefehl noch bearbeitet, berechnet die Adressberechnungseinheit "MCAR Compute" daraus die Adresse des nächsten Mikrobefehls im ROM und speichert das Ergebnis im `NextMCAR` Register

Der Wert von `NextMCAR` wird dann kurz vor Beginn des nächsten Taktes in das `MCAR` Register übertragen

Der nächste auszuführende Mikrobefehl wird dann aus dem ROM gelesen, liegt zu Beginn des nächsten Taktes vor und steuert die CPU

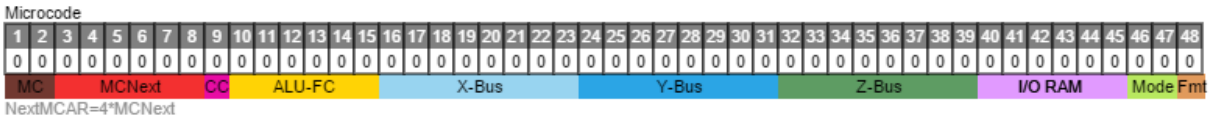
Für die Adressberechnungseinheit muss die Hardware der CPU um eine zusätzliche spezialisierte ALU erweitert werden. Sie wurde im CPU-Simulator aus Gründen der Übersichtlichkeit nicht explizit als ALU dargestellt.

# Sprünge im Mikrobefehlsspeicher

Es werden folgende Fälle unterschieden:

- Das Sprungziel steht von vornherein fest
- Das Sprungziel ergibt sich als Wert einer Berechnung mit einer speziellen ALU
- Das Sprungziel ergibt sich indirekt aus dem Inhalt des Speichers
- Das Sprungziel ergibt sich aufgrund einer Bedingung, die aus dem Flag-Register der CPU-ALU ablesbar ist

# Sprünge im Mikrobefehlsspeicher




Bits 1 bis 8 des Mikrobefehlworts bestimmen, welcher Befehl als nächster auszuführen ist

Bit 1 und 2 werden mit `MC` bezeichnet

- `MC`=00: absoluter Sprung: `NextMCAR = 4 * MCNext`
- `MC`=01: relativer Sprung vorwärts: `NextMCAR = MCAR + 1 + 4 * MCNext`
- `MC`=10: relativer Sprung rückwärts: `NextMCAR = MCAR + 1 - 4 * MCNext`
- `MC`=11: OpCode-gesteuerte und bedingte Sprünge: (später)

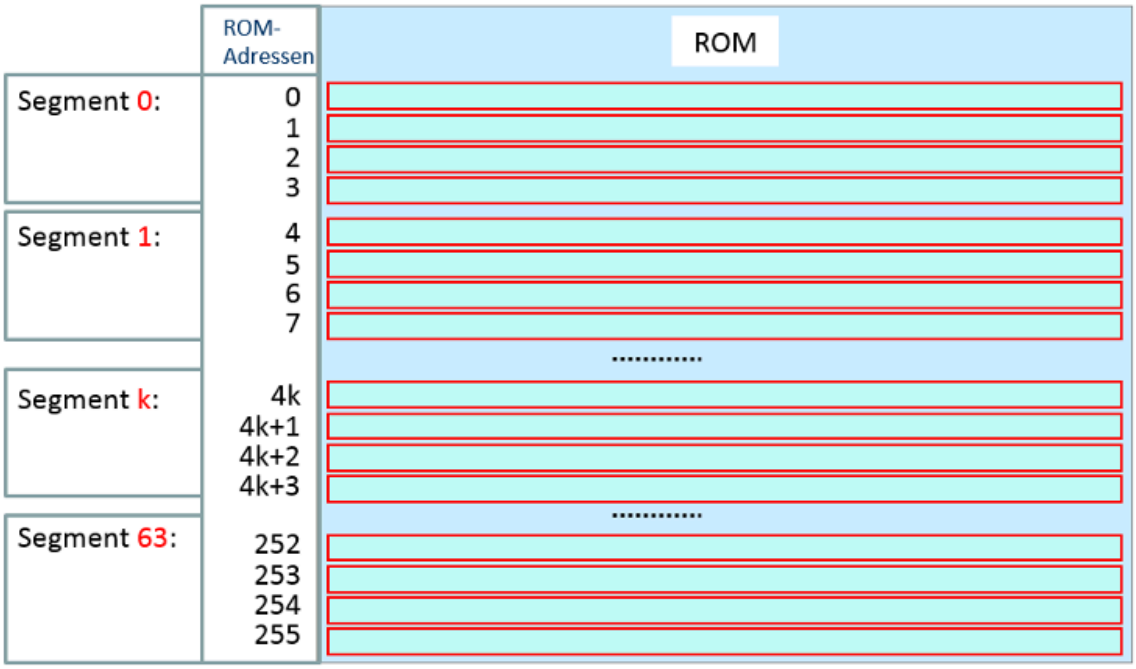
Bits 3 bis 8 werden mit `MCNext` bezeichnet und kodieren das Sprungziel

- Mit 6 Bits erhält man $2^6$=64 mögliche Werte
- Um diese etwas besser über den Speicher zu verteilen, wird `MCNext` mit 4 multipliziert
- Auf diese Weise ergibt sich eine logische Gruppierung von je 4 aufeinander folgenden Mikrocodeadressen zu einem Segment

# Absolute Sprünge

Je 4 ROM-Adressen werden zu einem Segment zusammengefasst

Absolute Sprünge (MC=00) sind nur an den Segmentanfang möglich



# Relative Sprünge

Im Sprungmodus `MC`=01 wird ein Vorwärtssprung relativ zum gegenwärtigen MCAR ausgeführt. Die Adresse des neuen Befehls ergibt sich aus:

```
NextMCAR = MCAR + 1 + 4 * MCNext
```

Im Sprungmodus `MC`=10 wird ein Rückwärtssprung relativ zum gegenwärtigen MCAR ausgeführt. Die Adresse des neuen Befehls ergibt sich aus:

```
NextMCAR = MCAR + 1 - 4 * MCNext
```

Würde in den obigen Fällen die +1 fehlen, so könnte man immer nur Befehle am Anfang eines Segments erreichen

Wenn `MCNext` = 0 ist, führen die beide relativen Sprungarten zum jeweils nächsten Befehl

Selbst ausprobieren:

- Automatisierung der Abarbeitung des vorherigen Beispiels `[5,6] := R1 + [5,6]`
- Im CPU-Simulator das Beispiel "Modify RAM with MCNext" auswählen

## OpCode-gesteuerte und bedingte Sprünge

Auch mit den bisher behandelten Möglichkeiten, Sprünge zu programmieren, ist das Mikroprogramm noch nicht von außen beeinflussbar

Diese Möglichkeit wird dadurch geschaffen, dass sich Sprünge im Mikrobefehlsspeicher von dem Inhalt des RAM Speichers oder von Ergebnissen von Operationen beeinflussen lassen

Beides ist im Sprungmodus `MC`=11 möglich

# OpCode-gesteuerte Sprünge

Computer werden durch Programme gesteuert, die aus Befehlen in Maschinensprache bestehen

Diese Maschinenbefehle befinden sich gemeinsam mit Daten im RAM

Jeder einzelne Maschinenbefehl besteht aus einem OpCode und ggf. weiteren Parametern

Ein Maschinenbefehl wird bei der Modell-CPU durch ein Mikroprogramm implementiert, das einige Mikrocodewörter umfassen kann und sich im Mikrobefehlsspeicher befindet

Der OpCode gibt, an wo sich dieser Befehl im Mikrobefehlsspeicher befindet

Daher werden OpCode-gesteuerte Sprünge im Mikrobefehlsspeicher benötigt

# OpCode-gesteuerte Sprünge

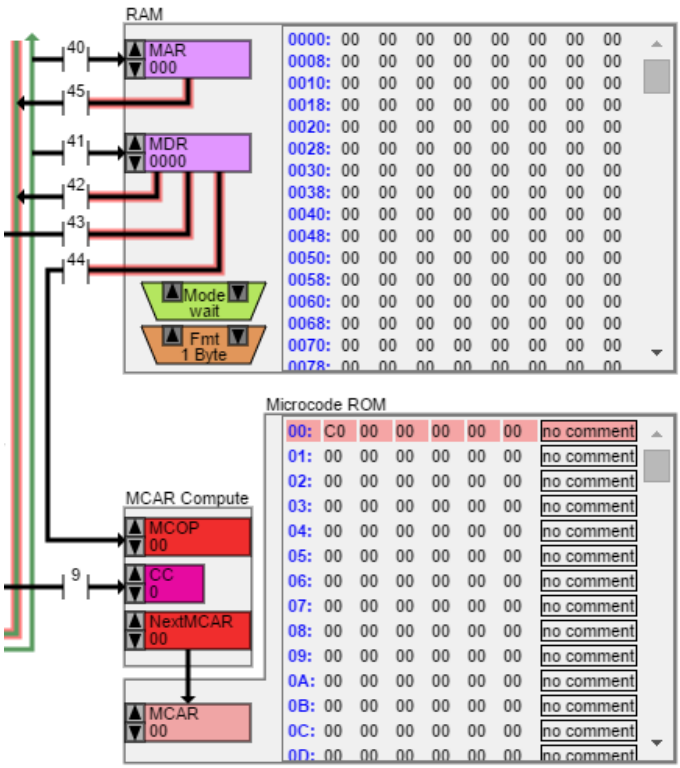
Für OpCode-gesteuerte Sprünge wird das `MCOP` Register verwendet, in das die Adresse des auszuführenden Sprungs von außen hineingeschrieben werden soll

Für diesen Zweck gibt es einen Datenpfad vom Datenregister des Speichers zum Register MCOP

Dieser Pfad kann über den `I/O RAM` Bus-Schalter $s_j = 44$ geöffnet oder geschlossen werden

Falls `MC`=11 ist und `MCNext` mit den Bits 00 beginnt, dann wird die Adresse für den nächsten Mikrobefehl so ermittelt:

```
NextMCAR = 4 * MCOP
```



# Bedingte Sprünge

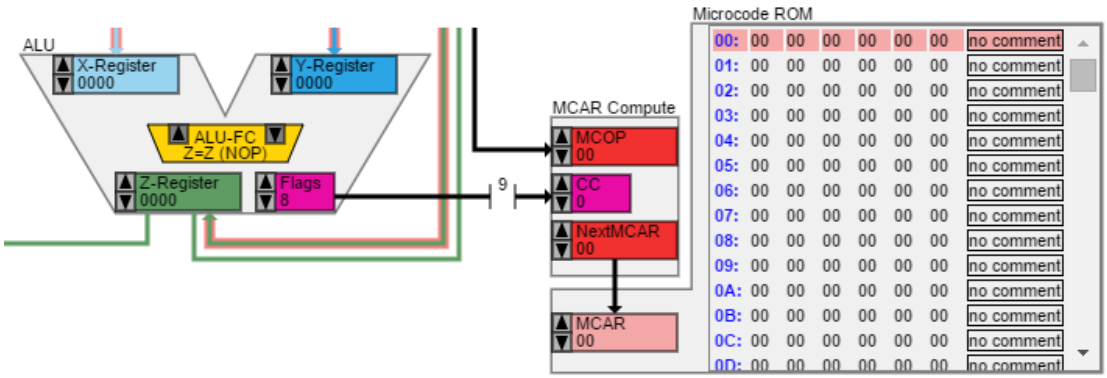
Letztendlich sollen auch Maschinenbefehle implementiert werden, die sich abhängig vom ALU Statusregister `Flags` anders verhalten

Dazu müssen bedingte Sprünge im Mikrocode ausgeführt werden, die sich auf den Inhalt des Statusregisters beziehen

Manchmal ist es erwünscht, die aktuellen Flags zu benutzen, manchmal die Flags aus einem vorangegangenen Befehl

Beides ist möglich. Dazu dient ein zusätzliches Condition-Code-Register `CC`

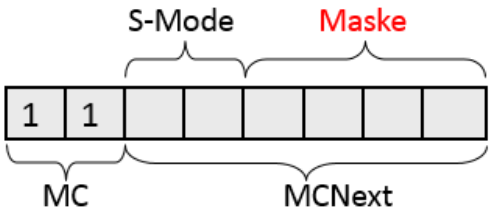
Nur wenn Bus-Schalter 9 gesetzt ist, wird der Inhalt von `Flags` nach `CC` übernommen



# Bedingte Sprünge

Die Bedingung für bedingte Sprünge wird mit Hilfe einer Maske ausgewertet

Falls `MC`=11 ist, wird MCNext in `S-Mode` und `Maske` zerlegt




Der Fall `S-Mode` = 00 liefert einen OpCode-gesteuerten Sprung und wurde bereits besprochen

Andernfalls bei `S-Mode` ≠ 00 werden die vier Bits des `CC`-Registers mit den vier Bits von `Maske` über ein logisches AND verknüpft

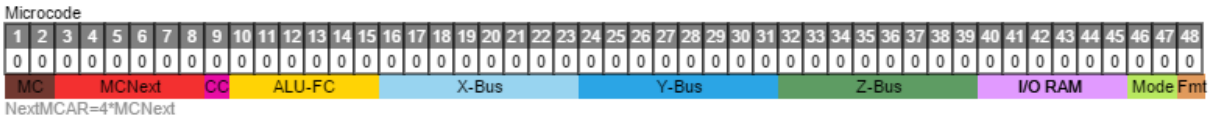
Ist das Ergebnis ungleich 0000, so wird der Sprung zur OpCode-Adresse ausgeführt:

```
NextMCAR = 4 * MCOP
```

Ist das Ergebnis gleich 0000, geht es mit dem nächsten Mikrobefehl weiter:

```
NextMCAR = MCAR + 1
```

# Zusammenfassung: Sprünge im Mikrobefehlsspeicher




| Sprungmode | Sprungziel |
|---|---|
| `MC=00` | `NextMCAR = 4 * MCNext` |
| `MC=01` | `NextMCAR = MCAR + 1 + 4 * MCNext` |
| `MC=10` | `NextMCAR = MCAR + 1 - 4 * MCNext` |
| `MC=11` | Zerlege `MCNext` in `S-Mode` (2 Bit) und `Maske` (4 Bit) |

Für Sprungmode `MC=11` gilt folgende Tabelle:

| `S-Mode` | Sprungziel |
|---|---|
| `S-Mode=00` | `NextMCAR = 4 * MCOP` |
| `S-Mode ≠ 00` | `NextMCAR = 4 * MCOP` falls `(Maske AND CC) ≠ 0000`<br>`NextMCAR = MCAR + 1` falls `(Maske AND CC) = 0000` |

# Mikroprogramm "Kleiner Gauss"

Als Beispiel soll ein Mikroprogramm erstellt werden, dass die Summe aller Zahlen von $1$ bis $N$ berechnet, wobei $N$ eine Zahl ist, die im RAM an der Stelle $(00)_{16}$ gespeichert ist. Das Ergebnis soll in Register $R_2$ geschrieben werden.

Laut Gaußscher Summenformel gilt:

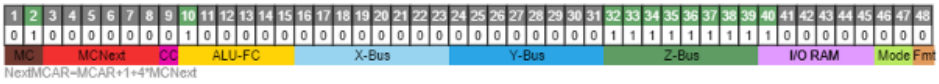
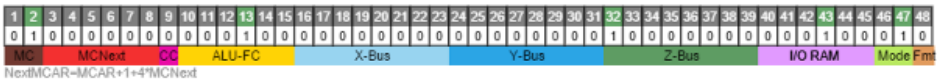
$$1+2+3+4+\cdots+N=\sum_{n=1}^{N} n = N\ \tfrac{N+1}{2} = \tfrac{N^2+N}{2}$$

allerdings soll diese Formel nicht verwendet werden, sondern die Summe schrittweise berechnet werden

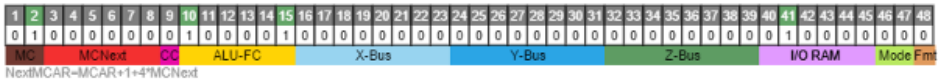
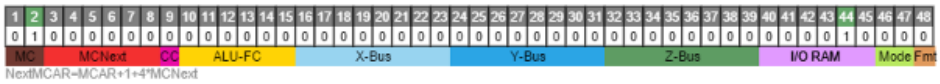
Wir erstellen das Mikroprogramm mit dem CPU-Simulator

- Dort können wir durch Anklicken die einzelnen Teile der Mikrobefehle zusammensetzen und diese mit Kommentaren versehen
- Oder einfach in der Combo-Box "Import example file" das Beispiel "Small Gauss: 1+2+3+...+n" auswählen

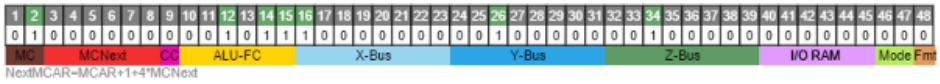
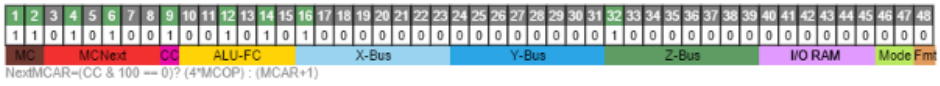
# Mikroprogramm "Kleiner Gauss"

| | | |
|---|---|---|
| 00: | Initialisiere $R_0, \ldots, R_7$ und MAR mit 0 | |
| | Phase 1 | keine Operation |
| | Phase 2 | `Z = 0` |
| | Phase 3 | `R1...R7 = 0, MAR = 0` |
| |  | |
| 01: | Lese $N$ aus [00H] und speichere $N$ in $R_0$ | |
| | Phase 1 | Speicher liest, `Y = MDR` |
| | Phase 2 | `Z = Y` |
| | Phase 3 | `Z = R0` |
| |  | |

# Mikroprogramm "Kleiner Gauss"

| | | |
|---|---|---|
| 02: | `MDR = 1` (Sprungvorbereitung) | |
| | Phase 1 | keine Operation |
| | Phase 2 | `Z = 1` |
| | Phase 3 | `MDR =` $Z_0$ |
| |  | |
| 03: | `MCOP = MDR` (noch Sprungvorbereitung) | |
| | Phase 1 | `MCOP = MDR` |
| | Phase 2 | keine Operation |
| | Phase 3 | keine Operation |
| |  | |

# Mikroprogramm "Kleiner Gauss"

| | | |
|---|---|---|
| 04: | Addiere $R_0$ zu $R_2$ | |
| | Phase 1 | `X = R0, Y = R2` |
| | Phase 2 | `Z = X + Y` |
| | Phase 3 | $R_2$ `= Z` |
| |  | |
| 05: | Dekrementiere $R_0$ und springe zu `4 * MCOP = 04`, falls Ergebnis > 0 | |
| | Phase 1 | `X = R0` |
| | Phase 2 | `Z = X - 1` |
| | Phase 3 | $R_0$ `= Z` |
| |  | |

# Vom Mikroprogramm zu Maschinenbefehlen

Die Vorstellung, größere Programme in Mikrocode programmieren zu müssen, ist abschreckend

Außerdem ist die CPU per Mikrocode-Programmierung nicht univeral einsetzbar, da der Inhalt des ROMs unveränderbar ist

Programmierer sollten sich nicht damit plagen müssen, Schalter in Datenwegen zu betätigen, Daten mühsam via Adress- und Datenregister aus dem Speicher zu lesen, Code-Adressen in Code-Adress-Register zu schreiben oder ähnliche lästige Dinge festzulegen

Die Details der Benutzung der Busse und der zeitlichen Abfolge der Teilschritte in den einzelnen Phasen sollen dem Programmierer ebenfalls verborgen bleiben

Eine abstraktere Programmierschnittstelle für die CPU: **Maschinenbefehle**

Diese abstrakte Sicht der CPU zeigt immer noch Register und RAM, verschwunden sind aber Busse, ALU, Adressrechner, Phasen und Bus-Schalter

Maschinenbefehle erlauben, Operationen direkt auf Registerinhalten durchzuführen und Daten zwischen Registern und RAM zu verschieben

Außerdem gibt es Maschinenbefehle, die direkte Sprünge zu besonders gekennzeichneten Code-Stellen bewirken, anstatt dass mühsam aus Sprungmode und Masken Programmverzweigungen hergestellt werden müssen

# Maschinensprache

Die Maschinensprache einer CPU ist die Menge von Maschinenbefehlen, die einem Programmierer zur Verfügung steht

Einige typischer Maschinenbefehle haben die Formen:

```
Op R1, R2
```

```
Op R1, W
```

```
Op R1, [Adresse]
```

Dabei ist

- `Op`: eine Operation
- `R`: ein Register
- `W`: ein konstanter Wert
- `[Adresse]`: eine Speicheradresse im RAM

# Maschinensprache

Beispiele:

| Maschinenbefehl | Bedeutung |
|---|---|
| `Add R1, R2` | `R1 = R1 + R2` |
| `Mov R1, R2` | `R1 = R2` |
| `Sub R1, 2` | `R1 = R1 - 2` |
| `Mov R1, 2` | `R1 = 2` |
| `Add R1, [61h]` | `R1 = R1 +` Inhalt von RAM Speicheradresse $(61)_{16}$ |
| `Mov R1, [61h]` | `R1 =` Inhalt von RAM Speicheradresse $(61)_{16}$ |
| `Mov [42h], R2` | Schreibe Inhalt von `R2` an RAM Speicheradresse $(42)_{16}$ |

# Sprünge in Maschinensprache

Ein absoluter Sprungbefehl lautet: `Jmp Adresse`
wobei `Adresse` eine RAM Speicheradresse ist, an der der nächste auszuführende Befehl beginnt

Bedingte Sprünge bestehen aus einer Vergleichsoperation mit anschließender♂Sprunganweisung, die auf dem Ergebnis des Vergleichs basiert

```
Op R1 , R2
CondJmp Adresse
```

- Jede Operation, die die Flags der ALU beeinflusst, kann als Vergleichsoperation dienen
- Die Sprungbedingung ergibt sich aus den Flags
- Meist werden die Flags mit einer Maske ♂ausgewertet. Masken für typische Sprünge sind meist durch Buchstaben codiert, z.B.:
  `JLE`: "Jump if less equal" ("Springe bei kleiner gleich")
- Beispiel:

```
Sub R1, R2
JLE 127
```

# Registerkonvention

Die Operanden von Maschinenbefehlen sind Register oder RAM Speicherplätze

Es dürfen aber nie beide Operanden RAM Speicherplätze sein

Die Register werden für bestimmte Zwecke reserviert, etwa als Programmzähler, als Zeiger auf den Stack oder den Datenbereich

Einige Register behält man als Rechenregister. Diese werden häufig Allzweckregister oder Akkumulatoren genannt. Viele Befehle der Maschinensprache sind nur mit Allzweckregistern durchführbar

Beispiel für eine Registerkonvention für unsere Modell-CPU:

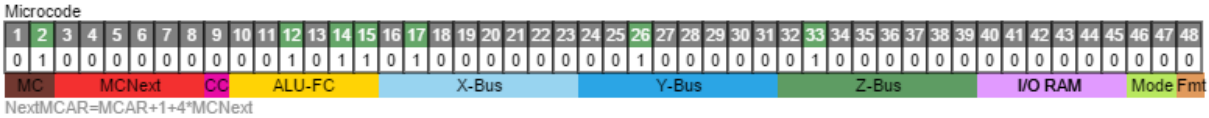
| Register | Vereinbarter Zweck |
|---|---|
| R0 | Programmzähler (Program Counter) PC |
| R1 | Akkumulator A |
| R2 | Akkumulator B |
| R3 | Loop Index I |
| R4 | Hilfsregister Aux |
| R5 | I/O Port |
| R6 | Data Pointer |
| R7 | Stack Pointer |

# Mikroprogrammierte Maschinenbefehle

Jeden Maschinensprachebefehl können wir durch ein kurzes Mikroprogramm implementieren

```
ADD A, B
```

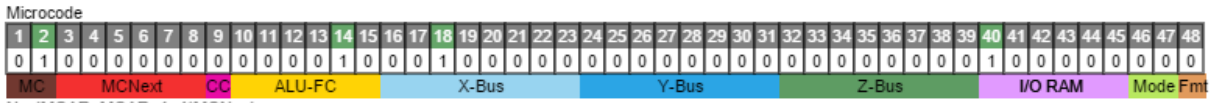
Microcode




NextMCAR=MCAR+1+4*MCNext

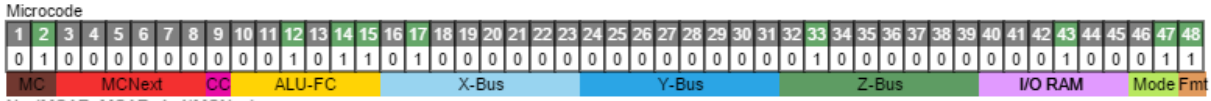
```
ADD A, [B]
```

Microcode




NextMCAR=MCAR+1+4*MCNext

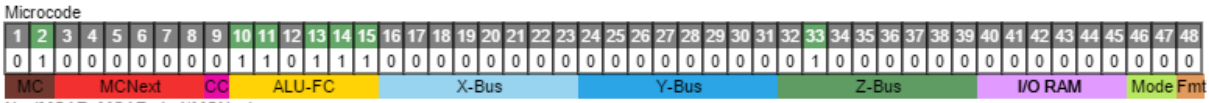
Microcode




NextMCAR=MCAR+1+4*MCNext

```
Mov A, 7
```

Microcode



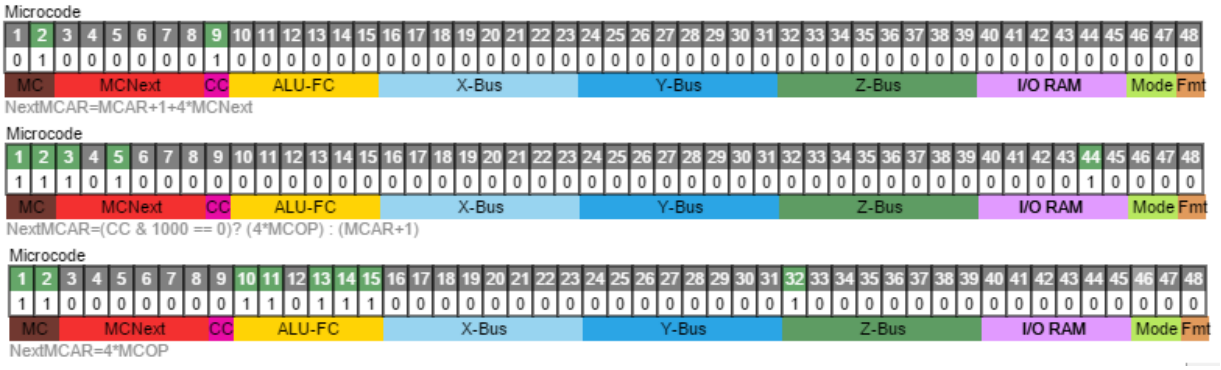

NextMCAR=MCAR+1+4*MCNext

# Mikroprogrammierte Maschinenbefehle

## Jump if not zero

Falls das Ergebnis der vorigen Operation = 0, führe den nächsten Maschinenbefehl aus (MCOP sei dafür entsprechend gesetzt), sonst setze `PC` (`R0`) auf 7 (Sprung!)

```
JNZ 7
```



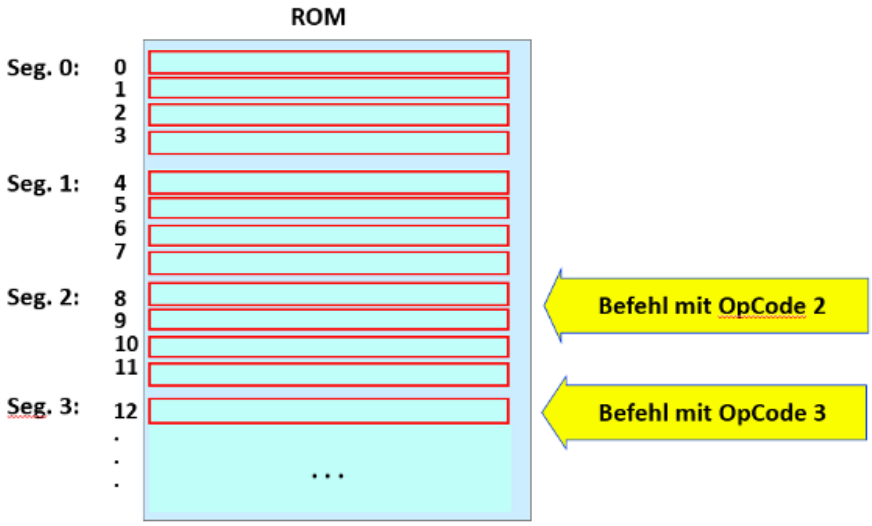
# Der Maschinensprache-Interpreter

Jeder Maschinensprachebefehl erhält eine Nummer, den OpCode

Er wird als kurze Mikrocoderoutine im ROM abgelegt

Die Routine beginnt immer am Anfang eines Segments

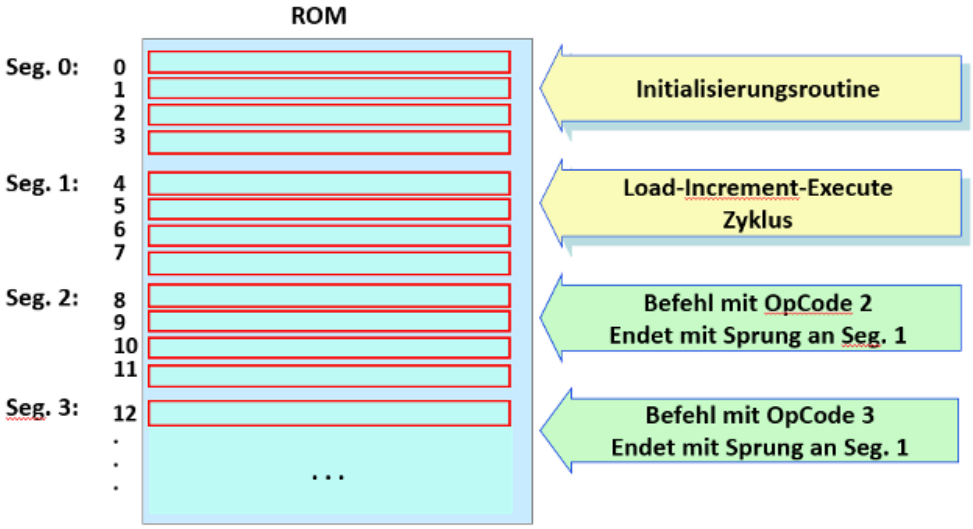
Der OpCode ist identisch mit der Segmentnummer



# Der Maschinensprache-Interpreter

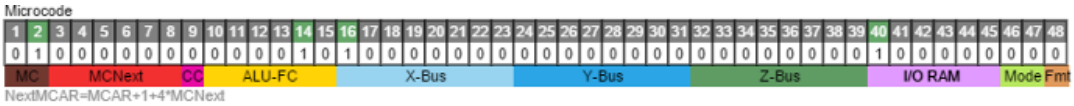
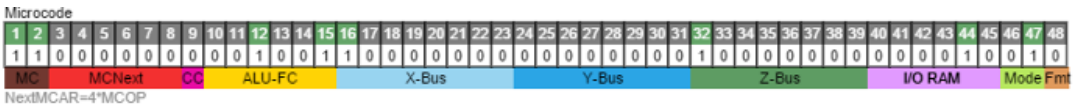
Ein Interpreter für Maschinensprache:

- Segment 0 enthält Initialisierungsbefehle
- Segment 1 enthält einen Load-Increment-Execute-Zyklus
- Jeder Maschinensprachebefehl endet mit einem Sprung an den Anfang von Segment 1



# Load-Increment-Execute-Zyklus

Der Load-Increment-Execute-Zyklus besteht aus 2 Mikrobefehlen:

| | | |
|---|---|---|
| 04: | Lade den Programmzähler | |
| | Phase 1 | `X = R0` |
| | Phase 2 | `Z = X` |
| | Phase 3 | `MAR = Z` |
| |  | |
| 05: | Jump Execute | |
| | Phase 1 | `MCOP = MDR` (Lies 1 Byte Opcode) |
| | Phase 2 | `Z = X + 1, MCAR = 4 * MCOP` |
| | Phase 3 | `R0 = Z, MAR = Z` |
| |  | |

# Load-Increment-Execute-Zyklus

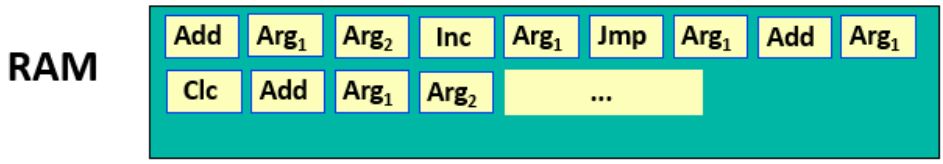
Achtung: Die Befehle des Load-Increment-Execute-Zyklus dürfen keine Flags verändern, da die Flags evtl. vom nächsten Maschinenbefehl benötigt werden

Das CC-Bit $s_j = 9$ muss also auf 0 gesetzt werden, damit die Befehle `Z = X` und `Z = X + 1` das CC-Register nicht beeinflussen können

# Maschinencode im RAM

Während das Mikroprogramm im ROM fest vorgegeben und nicht veränderbar ist, kann im RAM ein Programm in Form von Maschinenbefehlen zusammen mit seinen Argumenten liegen

Der Load-Increment-Execute-Zyklus führt dieses Programm aus




Jeder Maschinenbefehl, der Argumente benötigt, kann bei seiner Implementierung als Mikrocode davon ausgehen, dass MAR auf das erste benötigte Argument zeigt

Die Implementierung jedes Maschinenbefehls muss garantieren, dass hinterher der Programmzähler PC auf der Speicheradresse des Opcodes des nächsten Befehls im RAM steht

Selbst ausprobieren:
Im CPU-Simulator das Beispiel "Load-Increment-Execute cycle" auswählen

# Entwicklung eines Maschinenspracheprogramms

## 1. Schritt: Java-Program

```
// Kleiner Gauss: Berechne die Summe aller Zahlen von 1 bis N
static int gauss(int n) {
  int sum = 0;
  while (n > 0) {
   sum += n;
    n--;
  }
  return sum;
}
```

## 2. Schritt: Ersetze Programmstrukturen durch Sprünge (Linearisiere das Programm)

```
      int sum = 0;
loop: if(n == 0) goto end;
      sum = sum + n ;
      n = n - 1;
      goto loop
end:
}
```

# Entwicklung eines Maschinenspracheprogramms

2. Schritt: Ersetze Programmstrukturen durch Sprünge (Linearisiere das Programm)

```
      int sum = 0;
loop: if(n == 0) goto end;
      sum = sum + n ;
      n = n - 1;
      goto loop
end:
}
```

3. Schritt: Wähle Register oder Speicherplätze für die Variablen, benutze Assemblerbefehle

```
// Registerbelegung: sum --> A, n --> B
      MOV A, 0
loop: CMP B, 0
      JE end
      ADD A,B
      DEC B
      JMP loop
end:  STOP
```

# Entwicklung eines Maschinenspracheprogramms

4. Schritt: Implementiere die benötigten Befehle im Simulator

Für unser Beispiel sind es folgende 7 Befehle:

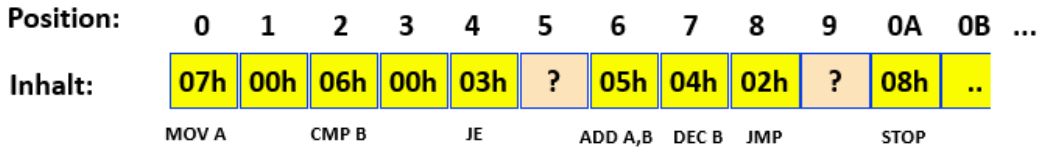
| Befehl | Länge im RAM | OpCode |
|---|---|---|
| `JMP <adr>` | 2 Byte | 02 |
| `JE <adr>` | 2 Byte | 03 |
| `DEC B` | 1 Byte | 04 |
| `ADD A,B` | 1 Byte | 05 |
| `CMP B, <num>` | 2 Byte | 06 |
| `MOV A, <num>` | 2 Byte | 07 |
| `STOP` | 1 Byte | 08 |

Dabei kann der OpCode frei zwischen 2 und 63 gewählt werden

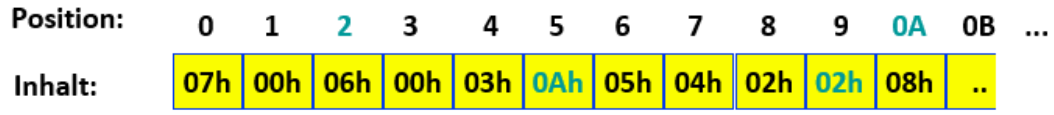
# Entwicklung eines Maschinenspracheprogramms

5. Schritt:

a) Übertrage das Programm in das RAM. Im ersten Durchlauf bleiben Sprungadressen noch offen




b) Trage die richtigen Sprungadressen durch "Backpatching" nach:




Selbst ausprobieren:
Im CPU-Simulator das Beispiel "Small Gauss (machine-code version)" auswählen

## Abstraktionsebenen

Im Laufe der bisherigen Vorlesungen haben wir mehrere aufeinander aufbauende Abstraktionsebenen kennen gelernt

- CMOS Transistor Schaltungen
- Logik Gatter
- ALU, Register, Bus-Schalter
- Mikrocode
- Maschinenbefehle

Diese Abstraktionsebenen sind wichtig, um sich als Hardware-Designer bzw. beim hardware-nahen Programmieren, auf die aktuelle Aufgabe zu fokussieren

# Gibt es Fragen?

Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: Kontakt

Weitere Vorlesungsfolien