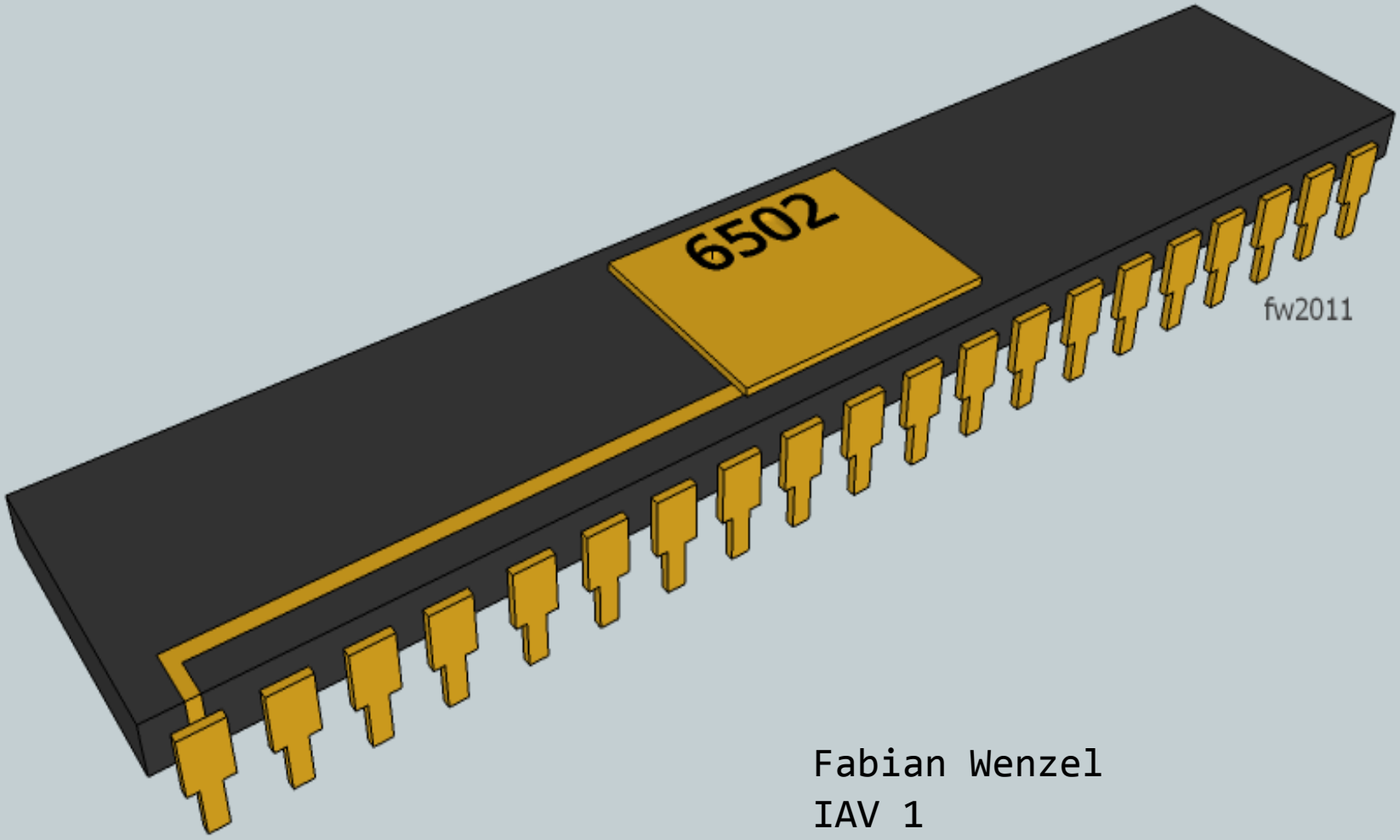
MOS Technologies 6502 MPU -

Der revolutionäre Mikroprozessor



Fabian Wenzel
IAV 1
Rudolf-Diesel-Fachschule
Nürnberg

Geschichtliches

- In den frühen 70er Jahren standen bereits leistungsfähige 8-Bit Mikroprozessoren zur Verfügung.
- Hohe Preise der etablierten 8-Bit Mikroprozessoren verhinderten jedoch eine weitere Verbreitung
  - Motorola MC 6800 : $300
  - Intel 8080 : $360
- Chuck Peddle, ein Elektroingenieur der bei Motorola in der Entwicklung des 6800 eingebunden war, wollte einen einfacheren und günstigeren Prozessor entwickeln.
- Motorola hatte kein Interesse an einem Low-Cost-Prozessor, weshalb Chuck Peddle schließlich mit vielen weiteren Mitarbeitern seines Teams zum damals noch unbedeutenden Halbleiterhersteller MOS Technologies wechselte, um dort eine günstigere Alternative zu entwickeln.

Die 650x - Familie

- Nur eine Versorgungsspannung von 5V notwendig
  - 8080 benötigt drei verschiedene Spannungen
- Buskompatibel zum 6800
  - 6501 kann einfach in bestehende Schaltungen gesteckt werden, keine Änderung der Hardware notwendig.
- Reduzierter Befehlssatz
  - wenige Befehle, aber viele Adressierungsarten und einfaches Programmiermodell
- Niedriger Verkaufspreis: 25 US-Dollar
  - weniger als ein Zehntel der Konkurrenz
  - Erschließung vollkommen neuer Märkte und Einsatzgebiete

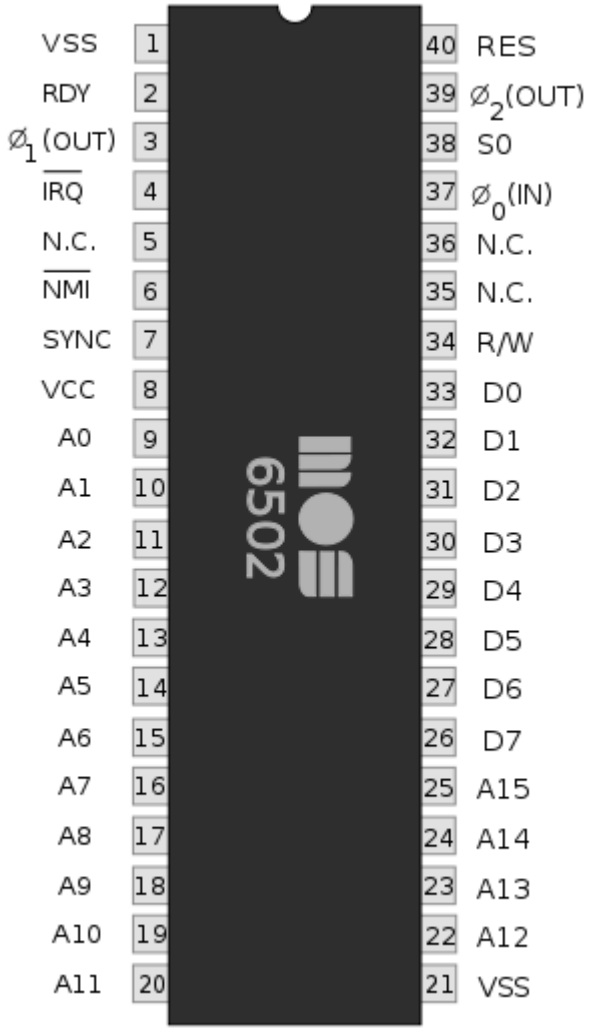
Reaktionen

- Deutliche Preissenkungen bei Intel und Motorola
- Klage wegen Patentverletzungen beim 6501
- Einstellung des 6501
- Entwicklung des 6502
- Einsatz des 6502 und seiner Derivate in den meisten Homecomputern und Spielkonsolen der späten 70er und frühen 80er Jahre, u.a.:
  - Apple I,II,III
  - CBM PET, VC 20, C64 (6510),C128 (8502), Plus4, C16
  - Atari 400, 800, VCS 2600 (6507)
  - Nintendo NES

Merkmale des 6502

- 1 Mhz Taktrate (bei Einführung)
- Zyklen/Befehl: ~3
- 64 KB Adressraum
- 8 bit Datenbreite, 8 bit ALU
- 13 Adressierungsarten
- 3 nutzbare 8-bit-Register
  - Akkumulator
  - X-index
  - Y-index
- 3 Interruptarten
  - IRQ
  - NMI – Non Maskable Interrupt
  - BRK – Software-Interrupt

Anschlussbelegung und Signale



Vcc Versorgungspannung 5V
Vss Masse

Phi 0 Takteingang
Phi 1 Taktausgang, interner Arbeitstakt
Phi 2 Taktausgang für externe Bausteine

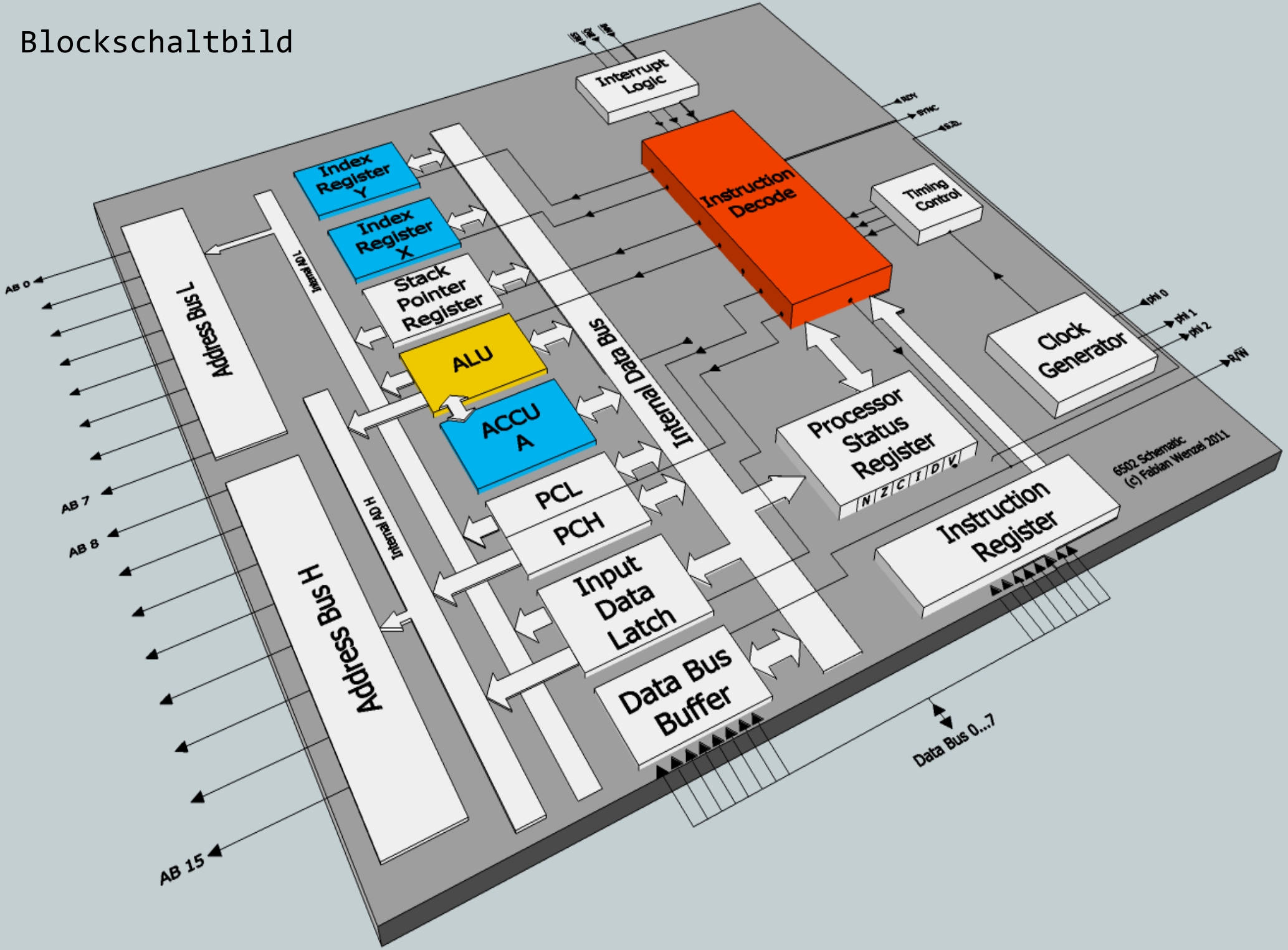
Phi 1,2 sind nicht überlappend, daher ist ein abwechselnder Buszugriff möglich.

A0 … A15 Adressbus, 16 bit
D0 … D8 Datenbus, 8 bit

Steuerbus

- RES Kaltstart (Sprung $FFFC)
- IRQ Unterbrechung (Sprung $FFFE)
- NMI Unterbrechung (Sprung $FFFA)
- R/W Steuerung Lese/Schreibzugriff
- S.O. Setzt Overflow-Flag
- RDY Ready, 0 hält Prozessor an
- SYNC Zeigt Zugriff auf Instruktion an
  - Durch Beschaltung von RDY/SYNC Einzelschrittbetrieb möglich!

Blockschaltbild



Register

- Akkumulator
  - 8 bit, direkt angebunden an ALU
    - Addition, Subtraktion, Vergleiche (ADC, SBC, CMP, BIT)
    - Multiplikation, Division um Faktor 2 (Shifting, Rotation – ASL, LSR, ROR, ROL)
    - Zugriff auf den Stack (Push, Pull - PHA, PLA)
    - Bit-Manipulation (AND, OR, EOR)

- Index Register X,Y
  - 8 bit, nutzbar für
    - Indizierten Speicherzugriff (LDA $1234, X , STA $1234, X)
    - Vergleiche (CPX)
    - Zähler mit einfacher Schrittweite (INX, INY)
    - Pointer-Zugriff (indirekt indiziert, z.B. LDA ($12), Y) - Nur Y-Register
    - Pointer-Tabellen-Zugriff (indiziert indirekt, z.B. LDA ($12, X)) - Nur X-Register
    - Stack-Manipulation - Nur X-Register

Register

- Program Counter (PC)
  - 16 bit, Speicheradresse der aktuellen Anweisung
- Stack Pointer (SP)
  - 8 bit, zeigt auf den oberstes Element des Stacks
- Flag Register
  - 8 bit, bestehend aus

Z	Zero Flag
N	Negative Flag - auch als S-Flag (signed) bezeichnet
C	Carry
V	Overflow Flag
I	Interrupt Disable Flag
B	Break Flag (Software-Interrupt)
D	Decimal Flag (BCD - Modus)

Flags

- Eine Instruktion (Mnemonic) hat im Allgemeinen zwei Ergebnisse:
  - Der neue Wert des Operanden
    - Im Speicher oder
    - Im Register (Akku, X, Y)
  - Der Zustand der ALU, ausgedrückt durch Flags
    - Im Processor Status Register (Flag-Register)
- Anhand der Flags werden Entscheidungen über den weiteren Programmablauf getroffen.
- Die Flags beeinflussen aber auch die Verarbeitung in der ALU (z.B. das Decimal Flag, das die ALU in den BCD-Modus versetzt)

Zero-Flag (Z)

- Zeigt an, daß Ergebnis der letzten Operation 0 war bzw. der gerade in ein Register geladener Wert 0 ist.
  - Gesetzt von
    - Ladebefehlen (LDA, LDX, LDY)
    - Vergleichsbefehlen (CMP, CPX, CPY)
    - Addition, Subtraktion (ADC, SBC)
    - Bitmanipulation (BIT, AND, OR, EOR)
  - Anwendung
    - Verzweigungsentscheidungen
      - BEQ (Branch if Equal – Flag gesetzt)
      - BNE (Branch if Not Equal – Flag gelöscht)

Negative-Flag (N oder S)

- Zeigt an, daß bit 7 gesetzt ist - z.B. weil das Ergebnis einer Rechenoperation negativ ist.
  - Gesetzt von
    - Addition, Subtraktion (ADC, SBC)
    - Bitmanipulation (BIT, AND, OR, EOR)
    - Zählbefehlen (INC, DEX, INY, DEY, INX, DEX)
  - Anwendung
    - Verzweigungsentscheidungen
      - BMI (Branch if MInus – Flag gesetzt)
      - BPL (Branch if Plus – Flag gelöscht)

Carry-Flag (C)

- Zeigt Überlauf oder Unterlauf der letzten Operation an
  - Gesetzt von
    - Addition, Subtraktion (ADC, SBC)
    - Bitmanipulation (ASL, LSR, ROL, ROR)
    - Vergleichsbefehlen (CMP, CPX, CPY)
    - Clear Carry, Set Carry (CLC, SEC)
  - Anwendung
    - Verzweigungsentscheidungen (16-bit-Arithmetik)
      - BCS (Branch if Carry Set – Flag gesetzt)
      - BCC (Branch if Carry Clear – Flag gelöscht)

Overflow-Flag (V)

- Zeigt an, daß bit 6 gesetzt ist
  - Gesetzt von
    - Addition, Subtraktion (ADC, SBC)
    - Bitmanipulation (BIT)
    - Hardware über S.O. Pin für schnelle I/O
    - Clear Overflow (CLV)
  - Anwendung
    - Verzweigungsentscheidungen
      - BVS (Branch if oVerflow Set – Flag gesetzt)
      - BVC (Branch if oVerflow Clear – Flag gelöscht)

Interrupt-Disable-Flag(I)

- Erlaubt oder unterdrückt Interrupt-Verarbeitung
  - Gesetzt von
    - Set Interrupt Disable Flag (SEI)
  - Gelöscht von
    - Clear Interrupt Disable Flag (CLI)
  - Anwendung
    - Zur Unterdrückung von IRQs, beispielsweise um eine Unterbrechung beim Einrichten einer neuen ISR (Interrupt ServiceRoutine) zu vermeiden.

BRK-Flag (B)

- Erkennung von Software-Interrupts
  - Gesetzt von
    - BRK-Befehl (löst Software-IRQ aus)
  - Anwendung
    - Unterscheidung von Soft- und Hardware IRQ in der ISR.
    - Das Flag wird sofort wieder gelöscht und existiert nur im Prozessor-Status, der beim IRQ auf den Stack gestellt wird und von der ISR ausgewertet wird.

Decimal-Flag (D)

- Versetzt die ALU in den BCD-Modus (Binary Coded Decimals)
  - Gesetzt von
    - SED – Befehl, CLD - Befehl
  - Anwendung
    - Dezimale Berechnungen
    - Ausgabe auf Sieben-Segment-Anzeigen.

Speicherorganisation

Bereich	Bezeichnung	Anwendung
0000-00FF	Zero-Page	Registerbank, schneller Zugriff
0100-01FF	Stack	Stapelspeicher für Rücksprung-Adressen oder Variablen
0200-FFF9	Freier Speicher	Frei für Programme, Variablen oder I/O
FFFA-FFFF	Vektoren	FFFA/FFFB - NMI-Vektor FFFC/FFFD - Reset-Vektor FFFE/FFFF - IRQ-Vektor

Die Zero-Page

- Die Zero-Page (0000 - 00FF) hat eine besondere Bedeutung. Der Zugriff auf diesen Speicher ist fast so schnell wie ein Registerzugriff, man kann sie daher als Pseudo-Registerbank bezeichnen.
- Dem Programmierer stehen daher praktisch 256 zusätzliche 8-Bit-Register oder, mit dem indirekten Zugriff, 128 16-Bit-Pointer zur Verfügung !

Der Stack

- Der Stack (0100 -01ff) wird vom Prozessor zur Ablage der Rücksprungadressen bei Funktionsaufrufen (JSR) oder auch vom Programmierer als Zwischenspeicher (PHA/PLA) verwendet.
- Durch die geringe feste Größe von nur 256 Bytes ist die theoretisch mögliche Rekursionstiefe auf 128 begrenzt.
- Ein geschickter Programmierer kann den Stack allerdings mit einer speziellen Systematik (Umkopieren und Manipulation des Stack-Pointers über TSX/TXS-Befehle) praktisch beliebig erweitern.

Die Sprungvektoren

- In den obersten 6 Bytes liegen die Vektoren für die Reset-, IRQ- und NMI-Routinen.
- Bei Eintritt eines dieser Ereignisse springt der Prozessor zu der im Vektor gespeicherten Adresse.
- Der Programmierer kann also beispielsweise einen Reset durch den Befehl

```
JMP $(FFFC)
```

  simulieren.

- Um einen IRQ/NMI zu simulieren, muss vor dem Aufruf des entsprechenden Vektors eine Rücksprungadresse sowie der gewünschte Status des Prozessors auf den Stack gelegt werden.

Befehlsübersicht

- Die Befehle können in 5 Gruppen eingeteilt werden:
  - Lade- und Speicheroperationen
  - Rechenoperationen
  - Stackmanipulation
  - Bitmanipulation
  - Sprünge
- Viele Operationen unterstützen verschiedene Adressierungsarten.
  Dies ist die größte Stärke des 6502.
- Je nach Adressierungsart sind die Befehle 1 bis 3 Bytes lang:

Accumulator Addressing

- Die Operation wird mit dem aktuellen Wert des Akkumulators ausgeführt. Kein Buszugriff notwendig.
- Der Befehl braucht nur 2 Taktzyklen !
- Verfügbar für ASL, LSR, ROR, ROL
- Beispiel:

```
ASL     ; shiftet Wert des Akkumulators einmal nach links
        ; entspricht Multiplikation mit 2
        ; Inhalt des MSB landet im Carry-Flag
```

Immediate Addressing

- Das Argument wird direkt im Code übergeben, d.h. unmittelbar nach dem Befehl (daher immediate) vom Speicher aus gesehen. Im Quellcode gekennzeichnet durch ein #.
- Der Befehl braucht nur trotz Buszugriff nur 2 Taktzyklen
- Verfügbar für
  - ADC,SBC
  - AND,EOR,OR
  - CMP,CPX,CPY
  - LDA,LDX,LDY
- Dies ist vergleichbar mit einer Konstante in C.
  Akku = 0x9 ;
- Beispiel:
  LDA #$09 ; lädt Akkumulator mit HEX 9

Absolute Addressing

- Das Argument wird aus einer 16-Bit-Speicheradresse gelesen, die im Code übergeben wird.
- 2 Buszugriffe für Adresse notwendig, daher typischerweise 4 Taktzyklen
- Befehle:
  - ADC,SBC
  - AND,BIT,EOR,ORA
  - CMP,CPX,CPY
  - ALS,LSR,ROL,ROR,DEC,INC (6 Zyklen, da lesen und schreiben)
  - LDA,LDX,LDY,STA,STX,STY
  - JMP,JSR
- Dies ist vergleichbar mit einer einfachen Variablen in C.

```
Akku = Speicher; // &Speicher = $1009
```

- Beispiel:

```
LDA $1009     ; lädt Akkumulator mit Wert aus $1009
```

Zero-Page Addressing

- Das Argument wird aus einer Speicheradresse in der Zero-Page gelesen, die im Code übergeben wird.

  1 Buszugriff für Adresse notwendig, daher typischerweise 3 Taktzyklen

- Befehle:
  - ADC,SBC
  - AND,BIT,EOR,ORA
  - CMP,CPX,CPY
  - ALS,LSR,ROL,ROR,DEC,INC (5 Zyklen, da lesen und schreiben)
  - LDA,LDX,LDY,STA,STX,STY
  - JMP,JSR
- Dies ist vergleichbar mit einer einfachen Variablen in C.

  Akku = Speicher; // &Speicher = $09
- Beispiel:

```
LDA $10  ; lädt Akkumulator mit Wert aus $10
```

Indexed Absolute Addressing, Indexed Zero Page Addressing

- Das Argument wird aus der angegeben Adresse + Offset gelesen. Der Offset wird einem der Index Register X oder Y entnommen.
- Trotz des erhöhten Aufwands brauchen die meisten Befehle nicht mehr Taktzyklen als bei nicht indiziertem Zugriff, also 4 Zyklen - allerdings auch für Zero-Page ! Hier werden technische Einschränkungen offenbar.
- Befehle:
  - ADC,SBC
  - AND,BIT,EOR,ORA
  - CMP,CPX,CPY
  - ALS,LSR,ROL,ROR,DEC,INC (5 Zyklen, da lesen und schreiben)
  - LDA,LDX,LDY,STA
  - JMP,JSR
- Vergleichbar mit einem indizierten Zugriff in C:

```
Akku = Speicher[x] ;
```

- Beispiel:

```
LDY #$17        ; lädt X-Register mit 17
LDA $1000,X     ; lädt Akkumulator mit Wert aus $1017
```

Indirect Indexed Addressing

- Die effektive Adresse wird aus der angegeben Adresse in der Zero Page. Auf den Inhalt der Adresse wird der Wert des Y-Registers addiert.
- Es werden mindestens 5 Zyklen gebraucht, da zuerst die Zieladresse aus dem Speicher gelesen werden muss (2x Buszugriff) und dann der Wert vom Bus gelesen werden muss.
- Befehle:
  - ADC,SBC
  - AND,BIT,EOR,ORA
  - CMP
  - LDA,STA (STA 6 Zyklen)
- Dies ist vergleichbar mit einem indizierten Pointer-Zugriff in C:

```
Akku = *(Speicher+y) ;
```

- Beispiel:

```
                        ; $10 = 20, $11 = 40
LDY #$23          ; lädt Y-Register mit 0x23
LDA ($10),Y             ; lädt Akkumulator mit Wert aus $4043
```

Indexed Indirect Addressing

- Die effektive Adresse wird aus der angegeben Adresse in der Zero Page + Wert des X-Registers entnommen.
- Es werden 6 Zyklen benötigt.
- Befehle:
  - ADC,SBC
  - AND,BIT,EOR,ORA
  - CMP
  - LDA,STA

- Dies ist vergleichbar mit einem indizierten Pointer-Zugriff in C:

```
Akku = *Speicher[x] ;
```

- Beispiel:

```
                    ; $12 = 10, $13 = 50
LDX #$02            ; lädt X-Register mit 0x2
LDA ($10,X)         ; lädt Akkumulator mit Wert aus $5010
```

Relative Addressing

- Auf die aktuelle Adresse (PC) wird der angegebene Wert addiert, bzw wird von derselben abgezogen.
- Verwendet bei bedingten Sprüngen (Branches), maximal 128 Byte rückwarts oder 127 Byte vorwärts
- 2 Zyklen, 3 Zyklen falls ein Sprung erfolgt.
- Befehle:
  - BCC,BCS
  - BEQ,BNE
  - BPL,BMI
  - BVC,BVS
- Keine direkte Entsprechung in C
- Beispiel:

```
LDX #$08 ; X mit 8 laden
DEX      ; X herunterzählen
BNE #$83 ; Solange x > 0, eine Anweisung zurückspringen
         ; (auf DEX-Befehl)
```

Indirect Addressing

- Die effektive Adresse wird der angebenen Speicheradresse entnommen.

- Verwendung nur beim Sprung-Befehl als Sprung-Vektor, beispielsweise für die IRQ- und Reset-Vektoren
- Befehl
  - JMP
- C: Funktionszeiger

- Beispiel:

```
                        ;       $0315 = 31, $0316 = EA
JMP ($0315)     ;       Sprung nach $EA31 (ISR beim C64)
```

Implicit Addressing

- Der Befehl betrifft direkt ein oder zwei Register oder den Stack
- Befehle:
  - Registerbefehle, 2 Zyklen
    - TAY,TYA,TAX,TXA,TSX,TXS
    - CLC,CLD,CLI,CLV, SEC, SED, SEI
    - DEX,DEY
  - Stackbefehle, 3-4 Zyklen
    - PHA,PLA,PHP,PLP
  - Sprungbefehle, 6-7 Zyklen
    - BRK,RTS,RTI

Lade- und Speicheroperationen

- LDA, LDX, LDY - LoaD Accu, LoaD X...
- Laden Wert aus einer Speicheradresse in das entsprechende Register
- Setzen Zero- und Negative-Flag entsprechend des Wertes
- STA, STX, STY
- Schreiben Wert des Registers in eine Speicheradresse

- Beispiel

```
                    ; $5000 = 3 ; $4005 = 16
LDY #$05            ;lädt Y-Register mit 5
LDA $4000,Y  ;lädt Akku mit Wert aus $(4000+5)
LDX $5000           ;lädt X-Register mit 3 (s.o)
STA $6000,X  ;schreibt Akku(16) in Zelle $6003
```

Transferoperationen

- TXA, TYA, TAX, TAY - Transfer X to Accu, Transfer Y to Accu...

  Der Wert eines Registers wird in ein anderes kopiert

- Manchmal notwendig zur Zwischenspeicherung, da nur der Akku eine Verbindung zum Stack hat (PHA,PLA, siehe dort.)

- Zero- und Negative-Flag werden entsprechend des Wertes gesetzt

- Beispiel : Wert des Y-Registers zwischenspeichern

```
....
TYA
PHA         ;  Akku auf Stack legen
JSR $2000 ;   Sprung in ein Unterprogramm, das den Wert des Y-
Registers
              möglichweise überschreibt
PLA      ;     Akku vom Stack holen
TAY      ;     Y-Register wiederhergestellt
.....
```

Rechenoperationen (Addition/Subtraktion)

- ADC - ADd with Carry, vorbereitet mit CLC - CLear Carry
- SBC - SuBtract with Carry, vorbereitet mit SEC - SEt Carry

- Sind immer auf den Wert des Akkus und eine weitere Speicherzelle bezogen. Das Ergebnis steht nach der OP im Akku.
- Setzen oder löschen das Carry-Flag, wenn Über- bzw. Unterlauf eintritt, sowie Negative- und Overflow-Flag entsprechend Bit 7 und 6 des Ergebnisses.
- Beispiel: 16-Bit addition der Zahlen in 6000/6001 und und 7000/7001, Ergebnis in 8000/8001: ($15F0 + $0220 = $1810)

```
CLC;          ; Carry löschen
LDA $6000     ; Lo-Byte   = $F0
ADC $7000     ;           = $20
STA $8000     ;           = $10, Überlauf, Carry gesetzt
LDA $6001     ; Hi-Byte   = $15
ADC $7001     ;           = $02 + Carry
STA $8001     ;  = $15 + $02 + 1 = 18;
```

Rechenoperationen (Zähler)

- INC, DEC
  Inkrementieren / Dekrementieren den Wert einer Speicherzelle
  Kein Einfluss auf Akku !
- INX, DEX , INY, DEY
  Inkrementieren / Dekrementieren den Wert des X- bzw Y-Registers
- Auswirkungen auf Zero- und Negative-Flag
- Verwendung typischerweise in Schleifen, zusammen mit den Vergleichsbefehlen:
- CMP, CPY, CPX
  Vergleichen das Register mit dem Wert der angegebenen Adresse durch Subtraktion und setzen/löschen entsprechend Zero, Negative- und Overflow-Flag
- Beispiel: Bereich von $4010 - 4020 nach $5010 kopieren

```
LDX #$10
LDA $4000,X
STA $5000,X
INX
CPX #$21
BNE #$89; Rücksprung um 9 Adressen auf LDA$4000,X
```

Rechenoperationen (Shifts)

- ASL - Arithmetic Shift Left, LSR - Logic Shift Right
  Der Wert im Akku oder in einer Adresse wird nach links oder rechts verschoben. Die freiwerdende Stelle wird mit einer 0 aufgefüllt, das hinausgeschobene Bit wandert in das Carry -Flag.

- ROL - ROtate Left, ROR - ROtate Right
  Der Wert im Akku oder in einer Adresse wird nach links oder rechts verschoben. Die freiwerdende Stelle wird mit dem Inhalt des Carry aufgefüllt, das hinausgeschobene Bit wandert wiederum in das Carry -Flag.

- Anwendungsmöglichkeiten:

- Multiplikation und Division, Wandlung von parallelelen Daten zu serieller Ausgabe, Bits zählen, z.B. für Parity-Check.

- Beispiel: Wert aus $2000 * 4 nehmen, $2000 = 5 = 00001001

```
LDA $2000      ; Akku = 5   = 00001001
ASL            ; Akku = 10 = 00010010
ASL            ; Akku = 20 = 00100100
STA $2000            ; $2000 = 20
```

Stackmanipulation

- PHA, PusH Accu
  PLA, PuLl Accu
  Der Wert des Akkus wird auf den Stack gelegt oder heruntergenommen.
  Verwendet zur Zwischenspeicherung oder für Funktionsparameter.

- PHP, PusH Processor Status
  PLP, PuLl Processor Status
  Der Wert des Flag-Registers wird auf den Stack gelegt oder heruntergenommen.
  Sinnvoll vor und nach Funktionsaufrufen, um den Zustand zwischenzuspeichern oder wiederherzustellen, sowie in Interrupt-Service-Routinen bzw. zum Debugging.

- TSX, Transfer Stackpointer to X
  TXS, Transfer X to Stackpointer
  Hierdurch kann der Stackpointer direkt manipuliert werden, um zum Beispiel die Größe des Stacks von 256 Bytes nahezu beliebig durch Umkopieren des Stacks zu erweitern.

Bitmanipulation

- AND, ORA, EOR (And, Or, Exclusive Or)

  Der Wert des Akkus wird dem Wert der angegebenen Adresse verknüpft und seinerseits wieder im Akku gespeichert.

  Zero- und Negative-Flag werden entsprechend gesetzt oder gelöscht.

- BIT

  Exotischer Befehl:
  Akku und Wert einer Adresse werden UND-verknüpft, Zero-Flag wird anhand des Ergebnisses gesetzt, Negative- und Overflow-Flag werden anhand des Wertes der Adresse gesetzt.

  Keine Rückwirkung auf den Akku !

- Beispiel:

```
LDA #$FF
AND #$0F
STA $2000 ; = $F0
```

Flagmanipulation

- SEC, CLC - SEt Carry, CLear Carry
  Carry-Flag setzen/löschen vor Rechenoperation
- SED, CLD - SEt Decimal, CLear Decimal
  Decimal-Flag setzen/löschen (für BCD-Modus)
- SEI,CLI - SEt IDF, CLear IDF
  Interrupt Disable Flag setzen/löschen, um IRQs zu sperren/zuzulassen
- CLV - Clear Overflow
  Overflow-Flag löschen - Das Overflow-Flag kann nicht explizit per Software gesetzt werden, sondern nur über oben erwähnte Signalleitung S.O., den BIT-Befehl oder ADC/SBC
- Flags können auch bewusst manipuliert werden, um einen relativen Sprung einzuleiten:
- Beispiel:

```
CLV           ;
BVC $4000     ; Spring IMMER nach $4000
```

Sprungbefehle (absolut, unbedingt)

- JMP - Jump
  Sprung zu Adresse, kein Rücksprung möglich.
  C-Entsprechung: goto
- JSR - Jump to SubRoutine
  Sprung zu Adresse, Rücksprungadresse wird auf den Stack gelegt.
  C-Entsprechung: function()
- RTS - Return from SubRoutine
  Sprung aus Routine, die mit JSR aufgerufen wurde, and die auf dem Stack abgelegte Adresse.
  C-Entsprechung: return
- BRK - Break
  Sprung zu Adresse, die Im Interrupt-Vektor abgelegt ist (Software-Interrupt). Gleichzeitig wird der Zustand der Flags auf dem Stack abgelegt.
- RTI - Return from Interrupt
  Sprung aus ISR an die auf dem Stack abgelegte Adresse.
  Gleichzeitig wird der Zustand der Flags auf den Zustand bei Eintreffen des Interrupts zurückgesetzt.

Sprungbefehle (relativ, bedingt)

- Verzweigung abhängig vom Zustand eines Flags
- Sprungweite maximal 128 Bytes rückwärts, 127 Bytes vorwärts
- Zero-Flag
  BEQ, Branch if EQual
  BNE, Branch if Not Equal
- Negative-Flag
  BMI, Branch if MInus
  BPL, Branch if PLus
- Carry-Flag
  BCS, Branch if Carry Set
  BCC, Branch if Carry Clear
- Overflow-Flag
  BVS, Branch if Overflow Set
  BVC, Branch if Overflow Clear

Sprungbefehle (Beispiel)

- Beispiel: Realisierung einer if/then/else Struktur

```
C:
void compare_x_to_5(int x)
{
   if (x < 5)
    { printf(„x kleiner 5“);}
   else if(x == 5)
          { printf(„x ist gleich 5“);}
          else
          { printf(„x ist größer 5“);}
   return;
}
```

-

```
compare_x_to_5:
          CPX #$05              ; CPX entspricht einer Subtraktion
          BMI    CALL_LESS      ; X -5  < 0 ?  -> Negative-Flag gesetzt (Bit7)
          BEQ    CALL_EQUAL     ; X- 5 == 0 ?  -> Zero-Flag gesetzt
          JSR   PRINT_GREATER_5
          RTS
CALL_LESS:
          JSR    PRINT_LESS_5
          RTS
CALL_EQUAL:
          JSR    PRINT_EQUAL_5
          RTS
```

-

```
Aufruf mit:       LDX #$x
          JSR compare_x_to_5:
```

...und zum guten Schluss

- NOP - No OPeration

  Auch ein Prozessor braucht mal eine Pause.

- 2 Taktzyklen
- Keine Argumente - keine Auswirkungen :)
- Wenn ohne Assembler, d.h. mit einem Monitorprogramm programmiert wird, zum Auskommentieren verwendet oder um Platz für eventuelle Ergänzungen zu reservieren
- Auch verwendet um Timing-Schleifen abzustimmen.

Praxis

- Nach dieser vielen Theorie möchte ich noch ein kleines praktisches Beispiel vorstellen.
  Der 6502 verfügt über keine Befehle zur Multiplikation und Division.

  Daher wollen wir hier eine kleine Routine dafür entwerfen.

- Ziel: Multiplikation von zwei positiven 8-Bit-Zahlen.
- Die Faktoren werden im Hauptspeicher an den Adressen $2006 und $2007 übergeben, das Ergebnis steht im Little-Endian-Format in $2000.
- Die Multiplikation kann durch wiederholte Addition realisiert werden.
- Wir werden die Multiplikation durch Addition und Verschieben implementieren...

Aufgabenstellung

- Ziel: Multiplikation von Zwei positiven 8-Bit-Zahlen.
- Die Faktoren werden im Hauptspeicher an den Adressen $2006 und $2007 übergeben, das Ergebnis steht im Little-Endian-Format in $2000.
- Die Multiplikation kann durch wiederholte Addition realisiert werden.
- Wir werden die Multiplikation durch Addition und Verschieben implementieren.
- Die Addition wird der Übersichtlichkeit halber in eine eigene Routine ausgelagert.
- Los gehts...

8-Bit-Additionsfunktion

- Zwei Zahlen werden addiert, indem man eine in den Akkumulator lädt und die andere mit dem ADC-Befehl (Add with Carry) addiert.
- Vor der Addition muss das Carry-Flag, also der Übertrag, gelöscht werden, da ansonsten das Ergebnis möglicherweise um eins verfälscht würde:

```
add_8:
    clc          ; Übertrag auf 0 setzen
    lda $2000    ; ersten Summanden laden
    adc $2002    ; zweiten Summanden addieren
    sta $2000    ; Ergebnis speichern, Übertrag steht im Carry
    rts;
```

16-Bit-Additionsfunktion

- Für die 16-Bit-Addition wiederholen wir den Vorgang einfach noch einmal mit dem Hi-Byte, ohne jedoch vorher den Übertrag zu löschen.
  Wir lassen das CLC also weg.

16-bit-addition: Summand A in $2000/1, Summand B in $2002/3 , Ergebnis in 2000/1

```
add_16:
   clc          ; übertrag löschen
   lda $2000    ; lo-byte
   adc $2002    ; lo-byte, carry enthät eventuellen überlauf
   sta $2000    ; lo-byte ergebnis

   lda $2001    ; hb
   adc $2003    ; hb + carry!
   sta $2001    ; hb ergebnis
   rts          ; return
```

Multiplikation

- Wir können die Multiplikation auf wiederholte Addition zurückführen.

  Ein Beispiel: 12 x 13 = 12 x 8
  + 12 x 4
  + 12 x 1

- Multiplikation x 2,4,8,16... können wir mit dem ASL (Arithmetic Shift Left) - Befehl realisieren.

- Wir nennen den ersten Faktor **Multiplikant**, den zweiten Faktor **Multiplikator**

- Wenn **Bit n** im **Multiplikator** gesetzt ist, addieren wir zum Ergebnis das Zwischenergebnis ((**Multiplikant** * **2^n**) = „summand“)

- Diesen Vorgang wiederholen wir für jedes Bit des Multiplikators.

- Um die Bits zu überprüfen, verschieben wir den Multiplikator einfach nach links oder rechts und prüfen dann das Carry Flag.

- Wir erhalten den

  Multiplikator in $2007
  Multiplikant in $2006

  Wir speichern das

  Ergebnis in $2000/1
  Summand in $2002/3

Multiplikation(1.Versuch)

```
MULTI_8X8:     ; Multiplikand in $2006, Multiplikator in $2007, Ergebnis $2000/1
     lda #$00  ;ergebnis löschen
     sta $2000
     sta $2001

     ldx #$08  ; 8 bits
NEXT_DIGIT:
     asl $2007      ; multiplikator einmal nach links shiften, msb -> carry
     bcc SKIP_ADD   ; wenn bit nicht gesetzt, keine addition nötig!
     lda $2006  ; multiplikant in zwischenspeicher
     sta $2002  ; lb summand
     lda #$00
     sta $2003  ; hb summand
     txa
     tay       ; y mal shiften
     cpy #$01   ; bei bit 0 muss nicht mehr geshiftet werden.
     beq SKIP_SHIFT
SHIFT:
     asl $2002  ; 0 -> lsb, msb-> carry
     rol $2003  ; carry -> lsb
     dey
     bne SHIFT  ;wiederholen solange y>0
SKIP_SHIFT:
     jsr add_16 ; das multiplikationsergebnis (2000/2001) mit dem
                ; eben berechneten zwischenergebnis addieren

SKIP_ADD:
     dex
     bne NEXT_DIGIT;
      rts
```

Ansatz:

Multiplikator von links nach rechts abarbeiten, Multiplikant jedesmal neu aufbauen.
Funktioniert FAST richtig, aber eben nur fast!
Nach einer halben Stunde vergeblicher Fehlersuche aufgegeben zugunsten eines anderen Verarbeitungs-modells, das ohne doppelte Schleife auskommt...
(nächste Seite)

Multiplikation(2.Anlauf)

```
MULTI_8X8:  ;Multiplikand in $2006, Multiplikator in $2007, Ergebnis $2000/1

    lda #$00       ;ergebnis löschen
    sta $2000
    sta $2001

INIT_MULTIPLIKANT:

    lda $2006      ; multiplikant in zwischenspeicher
    sta $2002      ; lb summand
    lda #$00
    sta $2003      ; hb summand

    ldx #$08       ; 8 bits

NEXT_DIGIT:

    lsr $2007      ; multiplikator halbieren durch rechtsshift
    bcc SKIP_ADD:  ; wenn bit nicht gesetzt ist, addierung einsparen.
ADD:
    jsr add_16

SKIP_ADD:
SHIFT:             ; summand verdoppeln durch linksshift
    asl $2002      ; 0     -> lsb, msb-> carry
    rol $2003      ; carry -> lsb

    dex            ; wiederholen, solange nicht bei letztem bit angekommen
    bne NEXT_DIGIT
    rts
```

Viel einfacherer Ansatz:

Von kleinster Stelle starten,
Multiplikant bei jeder Stelle verdoppeln und ggf. zum Ergebnis addieren.
Nur noch eine Schleife.

Diese Lösung funktioniert schon perfekt, ist aber noch nicht optimal.

Unabhängig von der Größe des Multiplikators wird 8 mal geschoben.

Multiplikation(3.Version)

```
MULTI_8X8: ; Multiplikant in $2006, Multiplikator in $2007, Ergebnis $2000/1

      lda #$00       ;ergebnis löschen
      sta $2000
      sta $2001

INIT_MULTIPLIKANT:

      lda $2006       ; multiplikant in zwischenspeicher 2002/3 schreiben
      sta $2002       ; lb summand
      lda #$00
      sta $2003       ; hb summand

NEXT_DIGIT:

      lsr $2007       ; binär halbieren durch rechtsshift
      pha             ; (*)
      bcc SKIP_ADD:   ; wenn bit nicht gesetzt ist, addierung
                      ; einsparen.
ADD:
      jsr add_16

SKIP_ADD:
SHIFT:                ; binaer verdoppeln durch linksshift
      asl $2002       ; 0     -> lsb, msb-> carry
      rol $2003       ; carry -> lsb

      pla             ; (**)  wiederholen, solange noch gesetzte
      bne NEXT_DIGIT ;  Bits im Multiplikator

      rts
```

Nur noch rechnen bis der Multiplikator abgearbeitet ist, d.h. 0 ist.

Der Multiplikator wird auf den Stack gelegt (*) und nach dem Shiften des Multiplikanten wieder hervorgeholt(**).
Ist der Multiplikator schon abgearbeitet, wird die Funktion beendet, dadurch ist die Laufzeit für kleine Multiplikatoren wesentlich geringer.

Keine Schleifenvariable mehr notwendig!

Multiplikation(Endstand)

```
MULTI_8X8: ; Multiplikant in $2006, Multiplikator in $2007, Ergebnis $2000/1

     ldy #$00         ; ergebnis auf 0
     sty $2000        ; ergebnis auf 0
     sty $2001        ; ergebnis auf 0

     lda $2007        ; multiplikator in akku laden
     beq EXIT         ; wenn multiplikator 0 ist, ergebnis auch 0 !

     sty $2003        ; hb summand auf 0

INIT_MULTIPLIKANT:

     ldy $2006        ; multiplikant in zwischenspeicher
     sty $2002        ; lb summand

NEXT_DIGIT:

     lsr              ; binär halbieren durch rechtsshift, kleinste
                      ; stelle -> carry
     tay              ; Multiplikator im Y-Reg. speichern
     bcc SKIP_ADD:    ; wenn carry(=bit) nicht gesetzt ist,
                      ; addierung einsparen.

ADD:
     jsr add_16       ; addiert 2000/1 und 2002/3, siehe dort.

SKIP_ADD:
SHIFT:                ; multiplikant verdoppeln durch linksshift
     asl $2002        ; 0        -> lsb, msb-> carry
     rol $2003        ; carry    -> lsb

     tya              ; Multiplikator aus Y-Register holen und
     bne NEXT_DIGIT   ; wiederholen, solange multiplikator > 0 ist
EXIT:
     rts
```

Der Multiplikator wird frühzeitig auf 0 überprüft, in diesem Fall erfolgt gar keine Berechnung.

Anstatt des Stacks wird das Y-Register zur Pufferung des Multiplikators benutzt, dadurch sind hierfür keine Buszugriffe nötig.

Wesentliche Verbesserungen könnten jetzt noch durch die direkte Einbindung der Addition erreicht werden, doch wir wollen uns hier zufriedengeben :)

Rückblick und Ausblick

Trotz seiner bescheidenen hier aufgezeigten Möglichkeiten hat der 6502 eine entscheidende Rolle bei der Etablierung des Personal Computers gespielt. Die Home-Computer-Revolution der späten 70er und 80 Jahre - ohne Apple II und C64 praktisch undenkbar.

Revolutionär war der 6502 keinesfalls in technischer Hinsicht, wohl aber in seinem Preis.

Mit ihm konnten erstmals für die Massen bezahlbare Mikro-Computer realisiert werden - sehr einfach, aber doch äußerst brauchbar.

man darf getrost annehmen, daß der Einstieg von IBM in den PC-Markt eine direkte Konsequenz des Home-Computer-Booms jener Zeit war.

Trotz - oder wegen ? - seiner kristallinen Einfacheit (nur 56 Befehle!) sind auf den 6502-basierten Maschinen alle möglichen Anwendungen ihrer Zeit realisiert worden -

wer weiß, was schon damals möglich gewesen wäre, hätte es schon den einfachen Austausch von Informationen über das Internet gegeben ?

Auch wenn die meisten 8-Bit Mikrocomputer mittlerweile entsorgt sein dürften, leben die Nachfahren des 6502 noch heute als Keyboard-Controller in vielen PC-Tastaturen oder - teilweise auf 16 Bit aufgebohrt - in Industrieanwendungen fort.

Wer etwas tiefer in die Materie einsteigen möchte, kann sich Emulatoren wie beispielsweise VICE aus dem Internet herunterladen, die meiste C64-Software ist heute frei verfügbar.

Und wer der Sache ganz tief auf den Grund gehen möchte, kann sogar unter Visual6502.org dem Prozessorkern live bei der Arbeit zuschauen.

Quellen und Bemerkungen

Die geschichtlichen Zusammenhänge und Preisangaben sind den deutschen und englischen Wikipedia-Seiten zum 6502, Chuck Peddle, dem 6800 und dem 8080 entnommen.

Die Informationen über die Opcodes sowie zum internen Aufbau entstammen einem Datenblatt der Firma Synertek.

Abbildung 6502 Pin-Out (c) Bill Bertram, verwendet unter den Bedingungen der Creative Commons Attribution 2.5 Generic Lizenz.

Zum Verständnis der Optimierungen ist ein Blick in eine Opcode-Tabelle des 6502 hilfreich, die hier aus Copyright-Gründen nicht beiliegt, aber zum Beispiel auf der Webseite http://ericclever.com/6500/ eingesehen und auch als PDF heruntergeladen werden kann.

Fabian Wenzel,

März 2011

(fabianwenzel@web.de)