Universität Bamberg




# Tutorial:
# Writing your own 6502 emulator

## Commodore 64 Symposium

## July 8th, 2024

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insb. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

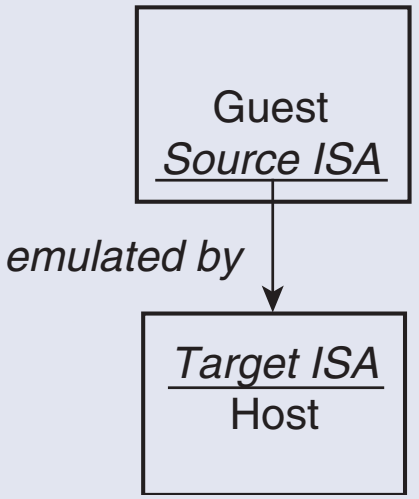
Licensed under CC BY-SA 4.0
unless noted otherwise



# Processor Emulation



- **CPU emulation is a central part of an emulator**
  - ***Instruction set emulation*** is a key aspect of many virtual machine implementations
  - The virtual machine must support a ***program binary*** compiled for an instruction set that is different from the one implemented by the host processor(s)
  - Emulation allows a machine implementing one instruction set, the target instruction set, ***to reproduce the behavior of software compiled to another instruction set***, the source instruction set
- We are writing an **interpreting emulator**



# Interpretation vs. Compilation



Translation from one (human) language to the other, e.g. from English to German, is a common problem.

In general, there are two ways to translate human languages:

- Translation while a person is speaking (translator or… interpreter!)
- Offline translation, e.g. of printed text




Both photos:
CC BY-NC-ND 2.0
European Union 2012 EP/
Pietro Naj-Oleari



We will use **interpretation** here – not an optimal way, but a great introduction to the topic

# CPU and System Emulation



**What do we have to emulate?**

- Internal **state** of the CPU: registers, flags
- **Instructions** of a CPU: semantics for each **opcode**

- **Memory:** can simply be an array of bytes
  - or multiple arrays, e.g. one for RAM and one for ROM

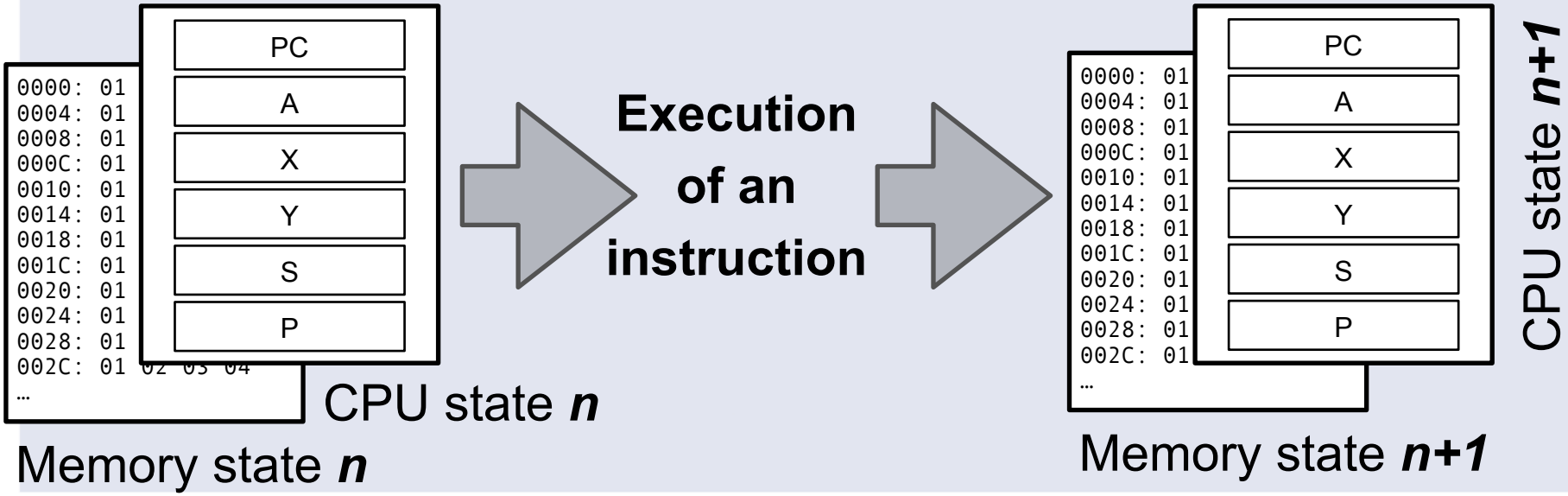
- **Input/Output:**
  - System calls: used when also emulating an operating system
    - e.g. x86 Linux binaries on an ARM Linux system
  - Peripherals: used when running "bare metal" code
    - like a whole operating systems or games for simple game consoles (e.g. C64, Gameboy, NES)

# Instruction Emulation



**Interpreting emulators work step-by-step**

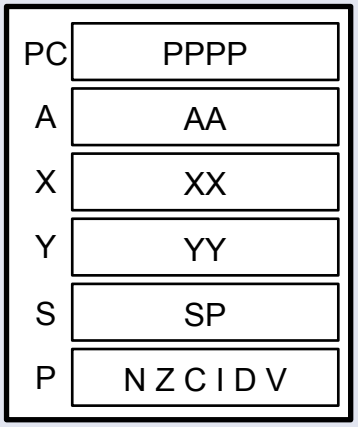
- Each interpreted machine instruction transforms an **initial state** of the processor and system into a **resulting state**
  - The state includes the CPU **registers** and **memory** (+ the state of I/O devices for complete system emulation)
- The specific transformation depends on the instruction



# CPU Emulation: Processor State



- **(Internal) state of the CPU:** registers and flags




**Program counter**, 16 bits

**Accumulator**, 8 bits

**Index Register X**, 8 bits

**Index Register Y**, 8 bits

**Stack Pointer S**, 8 bits, SP = 0x100 + S

**Processor Status P**, 7 bits

The **Processor Status** register holds flags that control the operation of the CPU (Interrupt enable I, BCD mode D) or indicate the result of operations (N=negative, Z=zero, V=overflow, C=carry)

**Meaning of the flags:**
- Z: result is value of all zero bits
- N: set sign bit in bit-position 7 present

Z and N are always updated whenever a value is transferred to a CPU register (A,X,Y) and as a result of any logical ALU operations. Z and N are also updated by increment and decrement ops memory.

C is used as a buffer/borrow in arithmetic operations. Any comparisons will update this additionally to the Z and N flags, as do shift and rotate operations

# CPU Emulation – Instructions





## What is an instruction?

- Computer science uses *abstraction* to provide a more simple, but possibly less detailed, view and realization of a problem
- **Programming in a high-level language, e.g. C**

```
int8_t x;
x = x + 42;
```

- **Programming in assembler: human-readable** (here: 6502)

```
lda loc_x   ; load var. x from memory to register a
adc #42     ; add value 42 to register a
sta loc_x   ; store value of register a to var. x
```

- **Machine code**

  `ad 00 10 69 2a 8d 00 10` (hexadecimal)

  **or** `10101101 00000000 00010000 01101001 00101010 1000…`

  - *For the CPU, machine instructions are indistinguishable from data – both are just bit patterns!*

# CPU Emulation – Addressing Modes



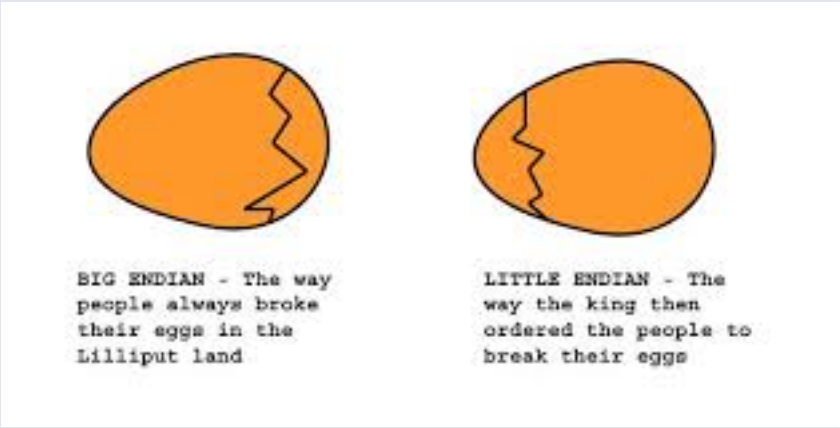
**Where does data for an instruction come from?**

- No or implied data – *implicit addressing* mode (no parameter)
  - e.g. CLC (clear carry), INX (increment X), ASL A (shift A left)
- Operand is byte after opcode – *immediate addressing* mode
  - e.g. LDA #42 (load 42 into A), EOR #23 (XOR of A and 23)
- Operand in memory – *absolute addressing* (operand is $HHLL)
  - e.g. STA 4711 (store A in memory), JMP 1234 (jump to 1234)
  - 16 bit values are stored in ***little endian byte order***
- Operand is relative offset in byte after opcode – *relative addr.*
  - e.g. BEQ dest (branch if equal) – only for branch instructions
  - The destination is a signed 2'd complement byte with values ranging from -128…+127, added to (address of branch + 2)
- Others: indirect, abs. X-/Y-indexed, zeropage (zp), zp X/Y-ind., …

# Endianness



Data that uses more than one byte (e.g. 16-bit addresses used in absolute addressing) can be stored in different ways: *least significant byte first* or *last*




**We call this "endianness" or "byte order"**

- Order in which the bytes of a data type using more than 8 bit are stored in memory
- Example:
  Decimal number 123456789 = hexadecimal **0x075BCD15**

**Little endian: least significant byte first**
(at lowest address)

00000000 **15**
00000001 **CD**
00000002 **5B**
00000003 **07**

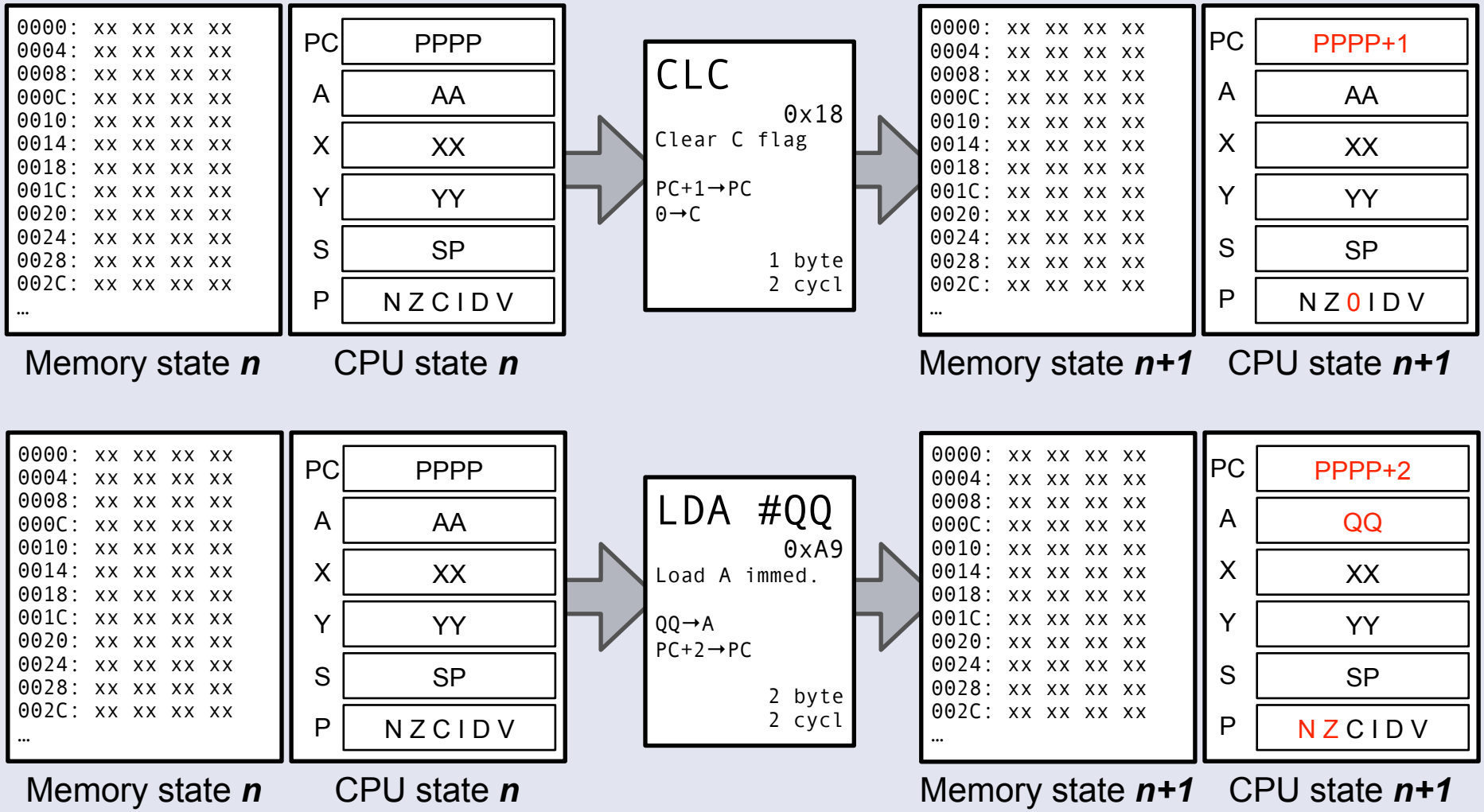
⇔

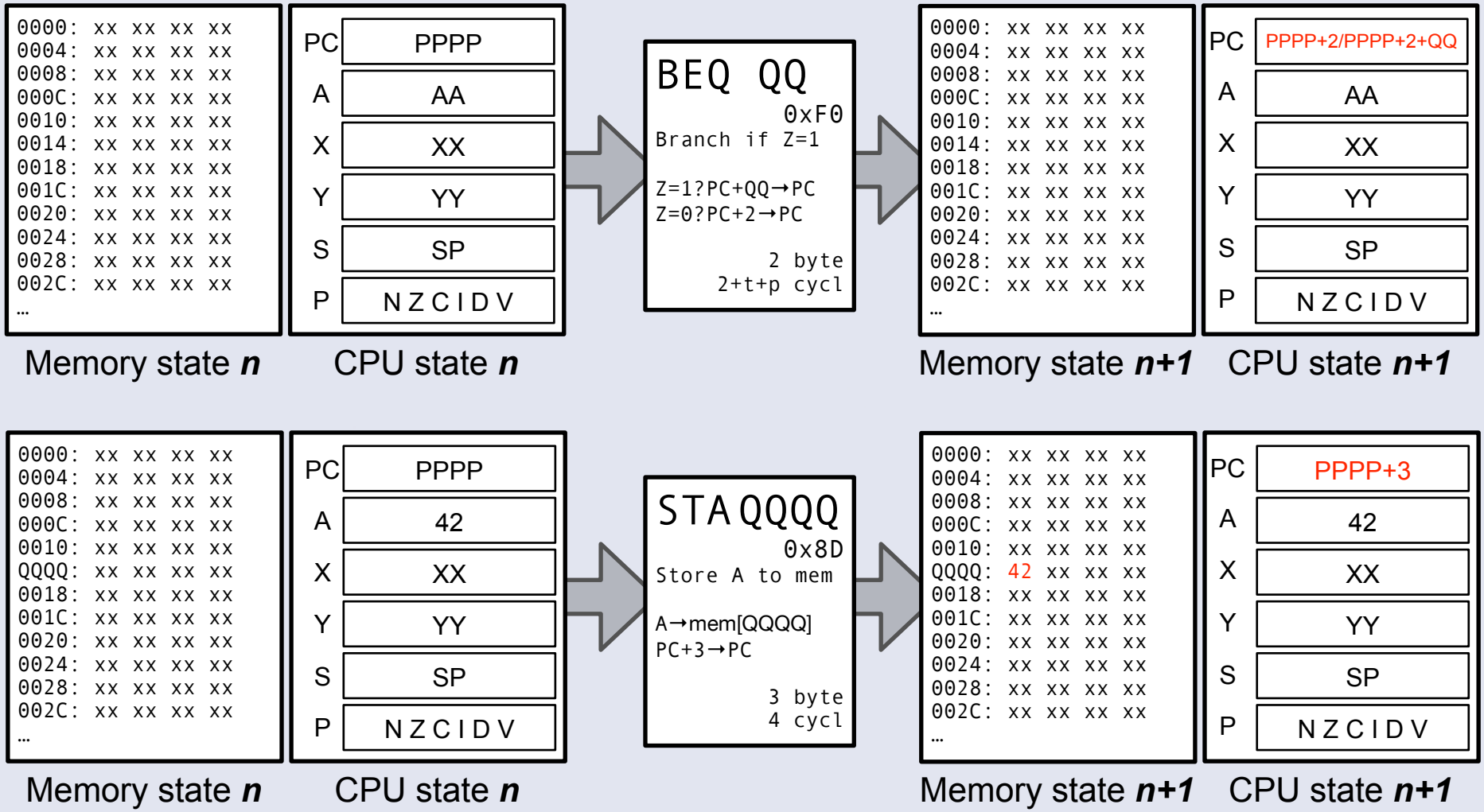
**Big Endian: most significant byte first:**

00000000 **07**
00000001 **5B**
00000002 **CD**
00000003 **15**

# Instruction Examples: CLC and LDA#

Universität Bamberg




0000: xx xx xx xx
0004: xx xx xx xx
0008: xx xx xx xx
000C: xx xx xx xx
0010: xx xx xx xx
0014: xx xx xx xx
0018: xx xx xx xx
001C: xx xx xx xx
0020: xx xx xx xx
0024: xx xx xx xx
0028: xx xx xx xx
002C: xx xx xx xx
…
PC PPPP
A AA
X XX
Y YY
S SP
P N Z C I D V
BEQ QQ
0xF0
Branch if Z=1
Z=1?PC+QQ→PC
Z=0?PC+2→PC
2 byte
2+t+p cycl
Memory state n
CPU state n
PC PPPP+2/PPPP+2+QQ
Memory state n+1
CPU state n+1
0000: xx xx xx xx
0004: xx xx xx xx
0008: xx xx xx xx
000C: xx xx xx xx
0010: xx xx xx xx
QQQQ: xx xx xx xx
0018: xx xx xx xx
001C: xx xx xx xx
0020: xx xx xx xx
0024: xx xx xx xx
0028: xx xx xx xx
002C: xx xx xx xx
…
PC PPPP
A 42
X XX
Y YY
S SP
P N Z C I D V
STA QQQQ
0x8D
Store A to mem
A→mem[QQQQ]
PC+3→PC
3 byte
4 cycl
Memory state n
CPU state n
0000: xx xx xx xx
0004: xx xx xx xx
0008: xx xx xx xx
000C: xx xx xx xx
0010: xx xx xx xx
QQQQ: 42 xx xx xx
0018: xx xx xx xx
001C: xx xx xx xx
0020: xx xx xx xx
0024: xx xx xx xx
0028: xx xx xx xx
002C: xx xx xx xx
…
PC PPPP+3
A 42
X XX
Y YY
S SP
P N Z C I D V
Memory state n+1
CPU state n+1

# Example Program: Assembler



- **Example program:** *output letters A..Z*
  - ASCII code values: `0x41..0x5A`

```
start:
  CLC             // clear carry flag
  LDA #0x41       // first value in A: 0x41 = 'A'
loop:
  STA 0xC000      // output to screen
  ADC #0x01       // increment value in A: next character
  CMP #0x5B       // last character reached?
  BCC loop        // no? execute next loop iteration
  BRK             // that's all
```

- **Can we execute this program directly?**
  - **NO** → have to translate to *binary representation* first ("assemble")

# Example Program: Machine Code



- **Example:** *output letters A..Z*
  - ASCII code values: 0x41..0x5A

Memory address | Opcode | Parameters (depend on instr.) | Source code and comments: labels, instructions *(not stored in binary program!)*

```
                    start:
0000 18               CLC           // clear carry flag
0001 A9 41            LDA #0x41     // first value in A: 0x41 = 'A'
                    loop:
0003 8D 00 C0         STA 0xC000    // output to screen
0006 69 01            ADC #0x01     // increment value in A
0008 C9 5B            CMP #0x5B     // last character reached?
000A 90 ??            BCC loop      // no? exec. next loop iteration
000C 00               BRK           // that's all
```

**Which value belongs here?!?**

**All numbers in hexadecimal!**

# Relative Jump Calculation

Universität Bamberg

```
                start:
0000 18           CLC           // clear carry flag
0001 A9 41        LDA #0x41     // first value in A: 0x41 = 'A'
                loop:
0003 8D 00 C0     STA 0xC000    // output to screen
0006 69 01        ADC #0x01     // increment value in A
0008 C9 5B        CMP #0x5B     // last character reached?
000A 90 ??        BCC loop      // no? exec. next loop iteration
000C 00           BRK           // that's all
```

**Which value belongs here?!?**

- **Problem:** *two byte addresses, but only one byte for address*
  - Address for branches is ***relative offset*** to current PC
  - Current PC = address of ***following instruction!*** (here: 0x10**C**)
- Calculate **2's complement difference:** 0x0003-0x000C = ???

```
  0x0003 = 0000 0000 0000 0011
- 0x000C = 0000 0000 0000 1100
           (1111 1111)1111 0111 = 0xF7 = -9 (2's complement)
```

# Assembling Our First Program

Universität Bamberg

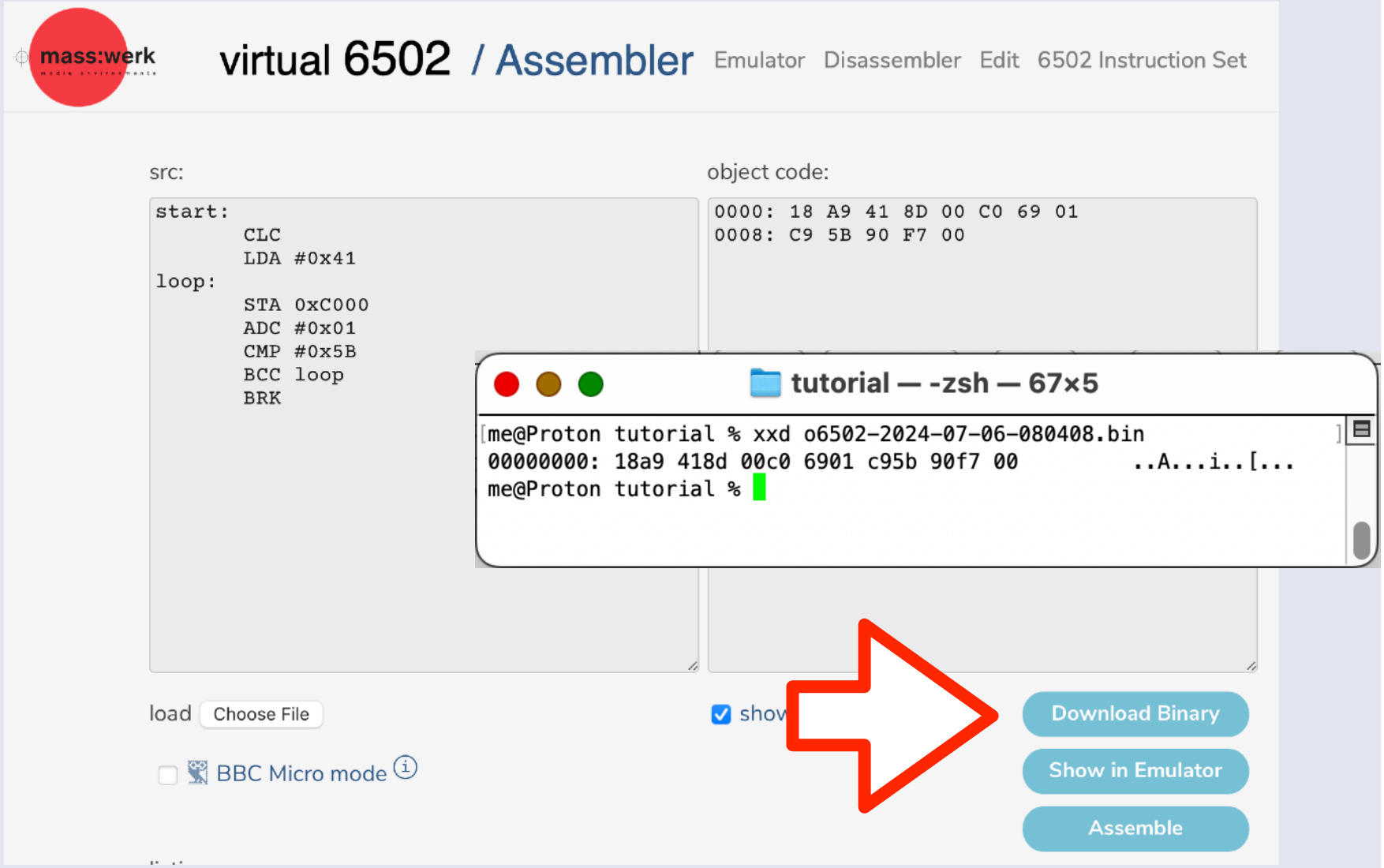
Convert assembler source code file into binary representation

https://www.masswerk.at/6502/assembler.html



# Assembling Our First Program – Result

Universität Bamberg



# Assembly Output Listing



Gives a view of the assembled instructions along with the addresses at which the instructions are located

listing:

```
pass 1

LINE  LOC          LABEL         PICT

   1  0000         START
   2  0000                       CLC
   3  0001                       LDA #0x41
   4  0003         LOOP
   5  0003                      STA 0xC000
   6  0006                       ADC #0x01
   7  0008                       CMP #0x5B
   8  000A                       BCC LOOP
   9  000C                       BRK

symbols
 LOOP                $03
 START               $00
                    $03
```

```
pass 2

LOC   CODE          LABEL         INSTRUCTION

0000                START
0000  18                          CLC
0001  A9 41                       LDA #$41
0003                LOOP
0003  8D 00 C0                   STA $C000
0006  69 01                       ADC #$01
0008  C9 5B                       CMP #$5B
000A  90 F7                       BCC $0003
000C  00                          BRK
```

# Assembly Output Listing



```
pass 2

LOC   CODE          LABEL         INSTRUCTION

0000                START
0000  18                         CLC
0001  A9 41                      LDA #$41
0003                LOOP
0003  8D 00 C0                  STA $C000
0006  69 01                      ADC #$01
0008  C9 5B                      CMP #$5B
000A  90 F7                      BCC $0003
000C  00                         BRK
```

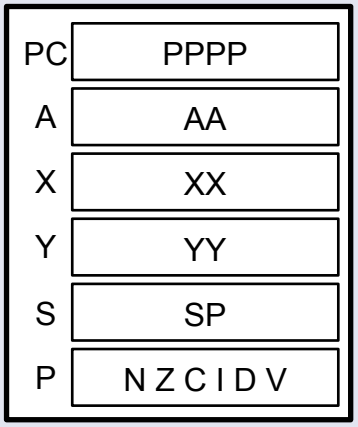
```
0100 18          start:
                   CLC           // clear carry flag
0101 A9 41         LDA #0x41     // first value in A: 0x41 = 'A'
                 loop:
0103 8D 00 C0      STA 0xC000    // output to screen
0106 69 01         ADC #0x01     // increment value in A
0108 C9 5B         CMP #0x5B     // last character reached?
010A 90 ??         BCC loop      // no? exec. next loop iteration
010C 00            BRK           // that's all
```

The assembler calculates the **relative branch offset** for us!

# CPU Emulation: Data Structures



- **Represent state of the CPU** (registers and flags) + **memory**




**Program counter**, 16 bits

**Accumulator**, 8 bits

**Index Register X**, 8 bits

**Index Register Y**, 8 bits

**Stack Pointer S**, 8 bits, SP = 0x100 + S

**Processor Status P**, 7 bits

```
uint16_t pc;          // program counter: 16 bit
uint8_t a,x,y,s       // A, Y, X, S are 8 bit registers
uint8_t p;            // N,V,-,B,D,I,Z,C: flag bits
enum { CF=0,ZF,IF,DF,BF,XX,VF,NF };       // flag bit numbers
uint8_t m[65536];       // memory: 64 kB RAM! :)
```

# Basic CPU Emulation: Init & Main Loop



- **Running code:** Big while loop + switch statement
  - Where do we start? *reset vector* of the CPU (6502: `0xFFFC`)
  - `0xFFFC` contains *pointer* to the first instruction address on the 6502

```
pc = m[0xFFFC]+256*(uint16_t)m[0xFFFD];
     // fetch first instruction from address
     // stored at 0xFFFC: 2 bytes, little endian!
a = 0; x = 0; y = 0; s = 0; p = 0; // initialize internal 6502 CPU state

while (1) {
  uint8_t opcode = m[pc];  // fetch instruction opcode from memory
  switch (opcode) {
    case 0xAD: // LDA xxxx (absolute)
         …     // implement semantics here
         break;
    case 0x69: // ADC #xx (immediate)
         …     // implement semantics here
         break;
    default:
         // unknown opcode - print error message
  }
}
```

# CPU Emulation: Instructions

Universität Bamberg

```
switch (opcode) {
   case 0xAD: // LDA xxxx (load A absolute)
        // absolute memory address: 2 bytes following opcode
        uint16_t addr = m[pc+1] + 256*m[pc+2];
        // load A register from given memory location addr
        a = m[addr];
        // calculate the Z and N flags
        if (a == 0) set_z() else clr_z();
        if ((a & 0x80) != 0) set_n() else clr_n();
        pc = pc + 3; // instruction is 3 bytes long
        break;
   case 0x69: // ADC #xx (add to A immediate with carry)
        // parameter is 1 byte directly following opcode
        // load parameter from mem[pc+1] and add to A
        a = a + m[pc+1];
        // ADC also adds the value of the carry flag (0/1)
        a = a + get_c();
        // calculate the Z and N flags
        if (a == 0) set_z() else clr_z();
        if ((a & 0x80) != 0) set_n() else clr_n();
        // TODO: also calculate C and V flags!
        pc = pc + 2; // this instruction has only 2 bytes
        break;
   // … and 149 cases more for all defined opcodes!
   default:          // catch the undefined ones here…
}
```

# Simple I/O Emulation



- **Example:** *memory-mapped output*
  - Print ASCII characters stored at address `0xC000` to screen
  - For a C64 emulator, we could e.g. print characters to the screen starting at memory address 0x0400

- This requires **special cases**…
  - Replace direct memory array access by **r**ead/**w**rite functions for 8 and 16 bit values:
    - `r8(addr), w8(addr, val), r16(addr), w16(addr,val)`

```
void w8(uint16_t addr, uint8_t val) {
  if (addr == 0xC000) { // memory mapped I/O output address
    putchar(val);
  } else {              // ordinary write to memory
    m[addr] = val;
  }
}
```

# Loading 6502 Machine Code



- **Self-written emulator**
  - *Memory needs to be initialized with instructions*
  - Entering hex bytes of instructions in the emulator source code
    - …inflexible and tedious…

```
// Our program in hex numbers
uint8_t prog[] = {
  0x18, 0xA9, 0x41, 0x8D, 0x00, 0xC0, 0x69, 0x01,
  0xC9, 0x5B, 0x90, 0xF7, 0x00
};

// Copy program to address 0x0000 in our emulated memory
for (int i=0; i<sizeof(prog); i++) {
  m[i] = prog[i]; // or w8(i, prog[i]);
}

// Set reset vector to start executing code at 0 (little endian)
m[0xFFFC] = 0x00; m[0xFFFD] = 0x00; // or use w16…
```

# Loading 6502 Machine Code



- **Self-written emulator**
  - *Assembled bytes can be directly loaded from a file*
    - *generated by the online assembler*

```
// Open the binary file
int fd = open("o6502.bin", O_RDONLY);
if (fd < 0) { perror("open"); exit(1); } // Check problems

// Read the binary file into the memory array
read(fd, m, 65536);

// Close access to the binary file
close(fd);

// Set reset vector to start executing code at 0 (little end)
m[0xFFFC] = 0x00; m[0xFFFD] = 0x00; // or w16…
```

# Live Coding!



- **I will use macOS with clang/LLVM here**
  - This code should also work on Linux with gcc or clang/LLVM and Windows with WSL2

- **Today, we will:**
  - Declare variables to store the 6502 CPU and memory state
  - Load a binary file into the emulated memory
  - Write helper functions (r8/w8/r16/w16) for accessing memory
  - Write the main loop with opcode fetch and switch statement
  - Emulate the instructions + addr. modes required for our example:
    - CLC, LDA #imm, STA abs, ADC #imm, CMP #imm, BCC rel, BRK
  - Test our emulator… and…
  - Debug it! 😀

# Evaluation of the Basic Emulation



**We emulate 7 of 151 opcodes – enough for our simple example!**

However, this basic emulation is **slow …….** 

- **How many 6502 instructions are executed** (count)**?**
- **How many x86 (or Arm) instructions are executed** (guess)**?**
- **Why is it so slow?**
  - Each instruction is read from memory and decoded every time
    - *also in loops – for every iteration!*
  - Parameters are separately fetched from memory
  - Overhead of while loop and of switch statement
  - Overhead of PC updates
  - Flag updates even if the updated flags are never evaluated

# References




1. James E. Smith and Ravi Nair:
   *Virtual Machines: Versatile Platforms for Systems and Processes*
   Chapter 1, Morgan-Kaufman, ISBN: 1558609105, 2005
2. Commodore Semiconductor Group:
   NMOS 6500 Microprocessors
   http://archive.6502.org/datasheets/mos_6500_mpu_nov_1985.pdf
3. 6502 instruction set
   https://www.masswerk.at/6502/6502_instruction_set.html
4. 6502 online assembler
   https://www.masswerk.at/6502/assembler.html
5. Visual6502 simulator: http://visual6502.org
6. Eric Jonas and Konrad Paul Kording,
   *Could a Neuroscientist Understand a Microprocessor?*
   PLOS Computational Biology 13(1): e1005268,
   doi: 10.1371/journal.pcbi.1005268