

COMP310: Legacy Game Systems

1: Assembly language

Learning outcomes

- ▶ **Explain** the historical significance of the 6502 CPU
- ▶ **Identify** the key components of the 6502 architecture
- ▶ **Read** and **write** simple programs in 6502 assembly language

Module introduction

Why Legacy Game Systems?

- ▶ To better understand the **history** of videogame systems
- ▶ To study low-level **machine architecture** without diving straight into complex modern architectures
- ▶ To make something under **constraints**:
 - ▶ Inspires **creativity**
 - ▶ Requires creative **problem solving**
 - ▶ Makes you a **better programmer**

Assignment 1: Constrained Development Task

- ▶ Design and implement a **de-make** of a well-known game
- ▶ Your de-make must run on the **NES** and be written in **6502 assembly**
- ▶ **In week 3:** present a **proposal** and **Trello board**
- ▶ **Tip:** scope carefully! Assembly programming is hard — keep it simple

Assignment 2: Research Journal

- ▶ As a **group**, make the ultimate **NES programming wiki**
- ▶ Deadline is in just over **2 weeks!**

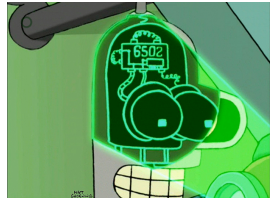
The 6502 CPU architecture

MOS Technology 6502



- ▶ Designed by Chuck Peddle and team at **MOS Technology**
- ▶ **8-bit** CPU, 16-bit addressing
- ▶ Clocked between **1MHz** and **3MHz**
- ▶ First produced in **1975**
- ▶ Still in production **today**

Uses of the 6502



Recap: hexadecimal notation

- ▶ We usually write numbers in **decimal** i.e. **base 10**

- ▶ Hexadecimal is **base 16**

- ▶ Uses extra digits:
 - ▶ A=10, B=11, ..., F=15

Hex	Dec	Hex	Dec	Hex	Dec
00	0	10	16	F0	240
01	1	11	17	F1	241
⋮	⋮	⋮	⋮	⋮	⋮
09	9	19	25	F9	249
0A	10	1A	26	FA	250
0B	11	1B	27	FB	251
0C	12	1C	28	FC	252
0D	13	1D	29	FD	253
0E	14	1E	30	FE	254
0F	15	1F	31	FF	255

How a CPU works

- ▶ Executes a series of **instructions** stored in **memory**
- ▶ An instruction is stored as an **opcode** followed by 0 or more **arguments**
- ▶ CPU has several **registers**, each storing a single value
- ▶ Instructions can read and write values in **registers** and in **memory**, and can perform **arithmetic and logical** operations on them
- ▶ The **program counter (PC)** register stores the address of the next instruction to execute

Registers on the 6502

- ▶ 6502 is an **8-bit** CPU, meaning each register stores **8 bits** of data
 - ▶ 1 byte
 - ▶ A number between 0 and 255
- ▶ A = **accumulator**
- ▶ X, Y = **index** registers
- ▶ SP = **stack pointer** register
- ▶ PC = **program counter** register (16 bits)
- ▶ **Status** register, composed of seven 1-bit **flags**

Assembly language

- ▶ Translates **directly** to machine code
- ▶ I.e. 1 line of assembly = 1 CPU instruction
- ▶ An **assembler** translates assembly to machine code
 - ▶ I.e. an assembler is a “compiler” for assembly language
- ▶ Each CPU architecture has its own instruction set therefore its own assembly language

Our first assembly program

```
LDA #$01  
STA $0200  
LDA #$05  
STA $0201  
LDA #$08  
STA $0202
```

Try it out! <http://skilldrick.github.io/easy6502/>

Our first assembly program

```
LDA #$01
```

- ▶ Store the value 01 (hexadecimal) into register A
- ▶ **LDA** (“load accumulator”) stores a value in register A
- ▶ # denotes a literal number (as opposed to a memory address)
- ▶ \$ denotes hexadecimal notation

Our first assembly program

STA \$0200

- ▶ Write the value of register A into memory address 0200 (hex)
- ▶ **STA** (“store accumulator”) copies the value of register A into main memory
- ▶ Note that address is a **16-bit** number (2 bytes, 4 hex digits)
- ▶ In this emulator the display is “memory mapped”, with 1 byte per pixel, starting from address 0200
 - ▶ This may **not** be the case on other 6502-based systems!

Assembly to machine code

```
LDA #$01
STA $0200
LDA #$05
STA $0201
LDA #$08
STA $0202
```

```
A9 01
8D 00 02
A9 05
8D 01 02
A9 08
8D 02 02
```

Note that the 6502 is **little endian**

- ▶ In 16-bit values, the “high” byte comes before the “low” byte
- ▶ Intel x86 is also little endian

Looping

- ▶ PC normally **advances** to the next instruction
- ▶ Some instructions **modify** the PC
- ▶ E.g. **JMP** (jump) sets the PC to the specified address

```
INC $0200 ; add 1 to the value at address 0200  
JMP $0600 ; jump back to beginning of program
```

- ▶ In this emulator the program always starts at address 0600
 - ▶ This may **not** be the case on other 6502-based systems!

Labels

- ▶ Don't use explicit jump locations in your code, it's not maintainable!
- ▶ Can add a **label** to a line of code, by giving a name followed by a colon
- ▶ Labels can then be used in instructions

```
start:  
    INC $0200  
    JMP start
```

- ▶ `start` is essentially a constant with value `$0600`
- ▶ The assembled code is exactly the same as for the previous slide

Conditional branching

```
LDX #$08      ; set X=8
decrement:
DEX           ; subtract 1 from X
STX $0200     ; store X in top left pixel
CPX #$03      ; compare X to 3
BNE decrement ; if not equal, jump
STX $0201     ; store X in next pixel
BRK           ; halt execution
```

$X = 8$

do

$X = X - 1$

memory[0200] = X

while $X \neq 3$

memory[0201] = X

Conditional branching

- ▶ Assembly language does not have **structured programming** constructs such as if/else, switch/case, for, while, etc.
- ▶ However all of these can be implemented using branch instructions
- ▶ ... which is exactly how compilers implement them

Subroutines

- ▶ **JSR** (jump to subroutine) works like **JMP**, but stores the current PC
- ▶ **RTS** (return from subroutine) jumps back to the instruction after the **JSR**
- ▶ These are used to implement **function calls**

Addressing modes

- ▶ Immediate: **LDA** #\$42
 - ▶ Load the literal value 42 (hex) into register A
- ▶ Absolute: **LDA** \$42
 - ▶ Load the value stored at memory address 42 (hex) into register A
- ▶ That # makes a big difference!
- ▶ Note that these actually assemble to **different** CPU instructions

Indexed addressing

```
LDA $0200,X
```

- Look up the value stored at memory address

$$0200 + (\text{value of X register})$$

and store it in A

- Can also do **LDA** \$0200,Y
- ... but **only** X and Y registers can be used for indexed addressing

Indexed addressing

```
    LDX #0          ; X=0  
loop:  
    TXA             ; A=X  
    STA $0200,X    ; store A to 0200+X  
    INX             ; X++  
    JMP loop        ; loop forever
```

- ▶ Why does it stop $\frac{1}{4}$ of the way down?
- ▶ Hint: it stops after filling 256 pixels...

Next steps

- ▶ **Work through** the tutorials on `http://skilldrick.github.io/easy6502/`
- ▶ **Understand** and try to **modify** the Snake game code