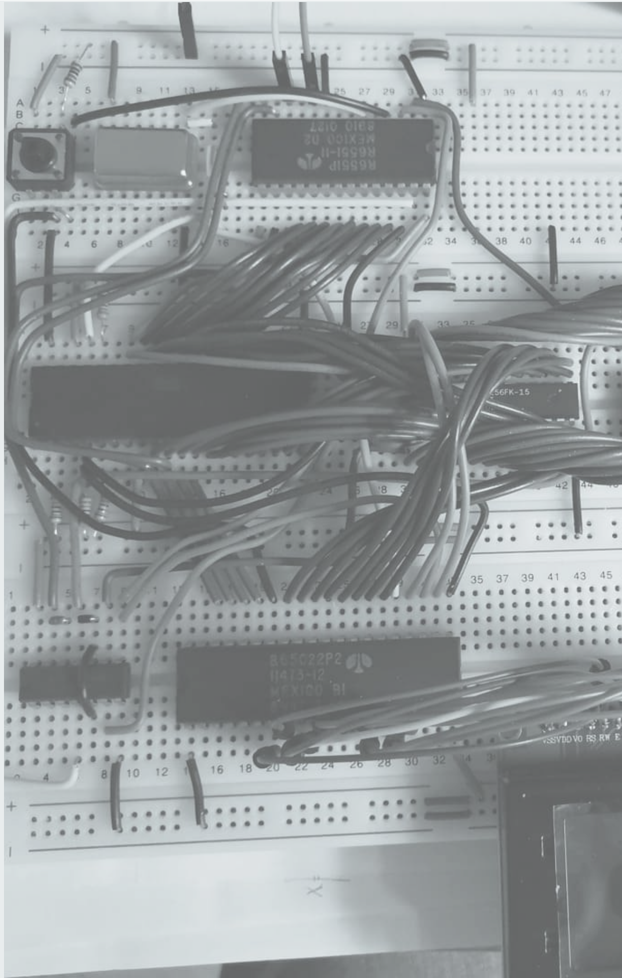
Implementing FORTH on my 6502 computer

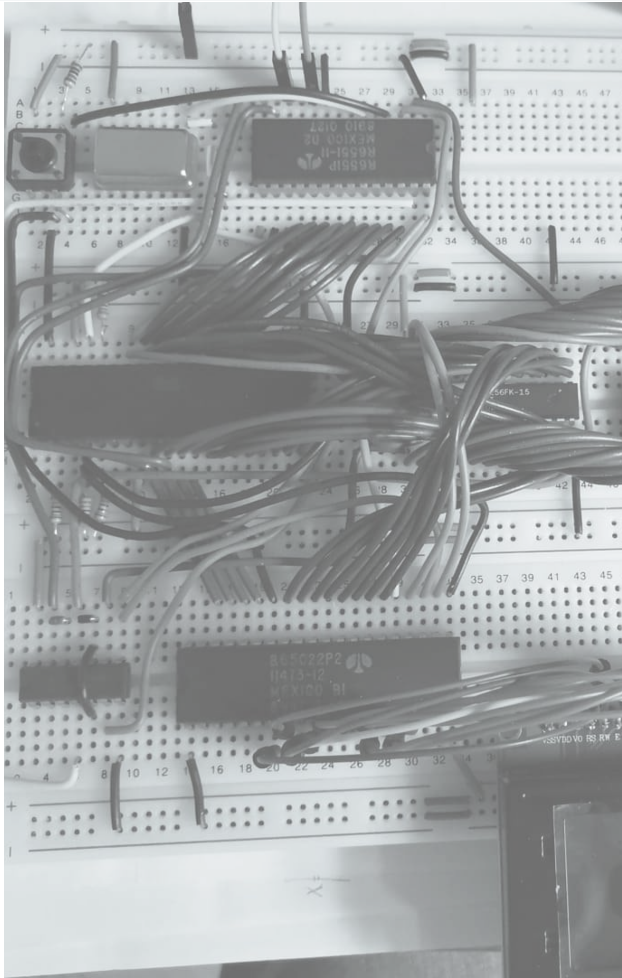
Alexandre Dumont
@adumont

#FORTH2020
Dec, 11th, 2021



Content

- Introduction
  - Motivation
  - My 6502 breadboard computer
- 6502 basics
  - Registers, ZP, Stack
  - Addressing Modes
- My FORTH Implementation
  - Dictionary, Inner interpreter, Data stack
  - Development methodology
  - Features & current limitations
- Quick Demo



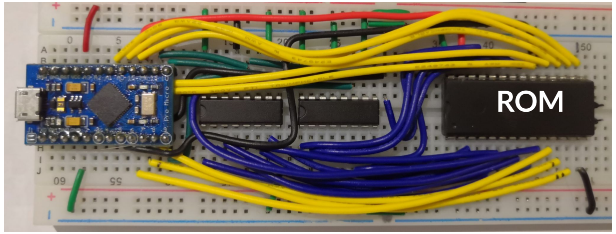
Motivation

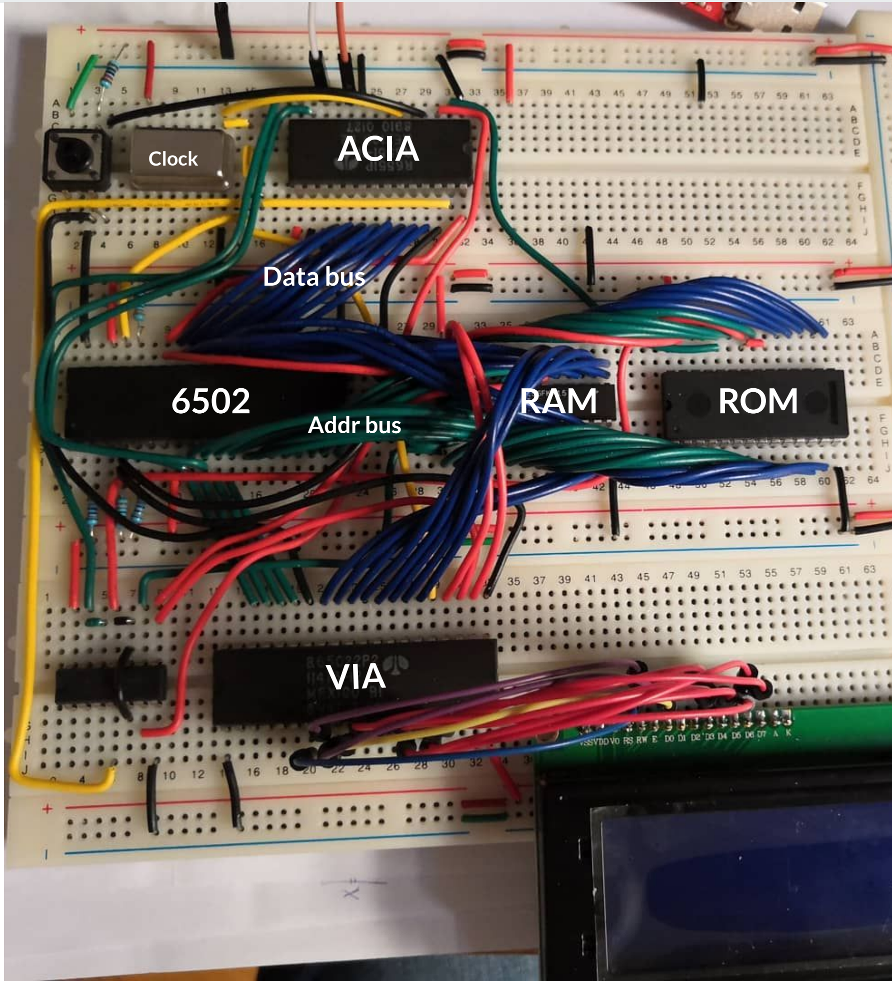
- Enjoy!
- Learn something new
  - 6502 assembly
  - FORTH internals
  - Eventually learn some FORTH
- Make my 6502 computer *usable*

My 6502 computer

- **WDC 65c02s**
- 1.8432 MHz crystal oscillator (clock)
- 32KB EEPROM, 16KB SRAM
- 6551 ACIA (serial port) ← User interface
- 6522 VIA (Versatile Interface Adapter) ≈ GPIO
- 20x4 LCD display
- No keyboard, no VGA display

- EEPROM Programmer:





A computer... in need of **some software**

- I began coding simple 6502 assembly programs:
  - From simple registers manipulation to writing a “Hello World” on the LCD display

- I developed my own **ROM monitor**:
  - Read/write to memory (or I/O)
  - Jump to code at a specified address
  - Return to monitor on BRK and then resume
  - Dump and Edit registers

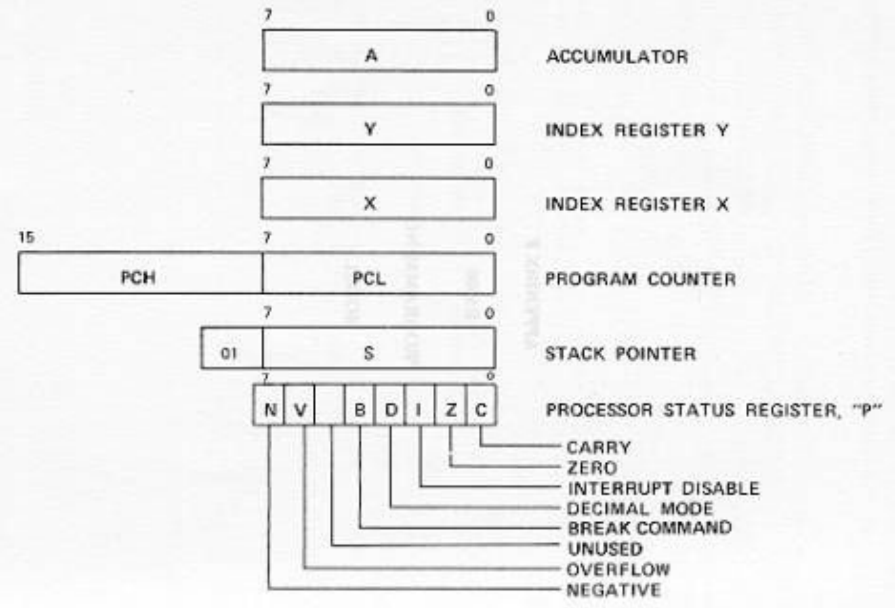
Around that time (April 2021) is when I then discovered the **FORTH** language

6502 basics*

*Relevant to *my FORTH implementation* ;-)

6502 Registers

- 8 bits CPU, 16 bits wide address bus
- It can address 64KB of memory space (RAM+ROM+I/O devices)
- 8 bits **registers**:
  - A: Accumulator: general purpose, ALU
  - X, Y: indexes, used in addressing modes
  - S: Stack pointer ($01**xx**)
  - **P**: processor flags
- **PC**: program counter (16 bits)
- Zero Page ($00xx): 1 byte address! Can act as 16 bits registers in complex addressing modes



6502 *hardware* stack

- Resides in memory page 1: `$0100:$01FF`

- Register **S** is a pointer into the stack
  - $0100+S is the next available location in the stack memory area

- The stack is used by the 6502:
  - Calls to subroutines (store return address)
  - Responding to interrupts (store status register and return address)

- Instructions to push/pop 8 bit registers on/off the stack:
  - **pha / pla**
  - **phx / plx, phy / ply** (*)
  - php / plp

* 65c02 specific instructions

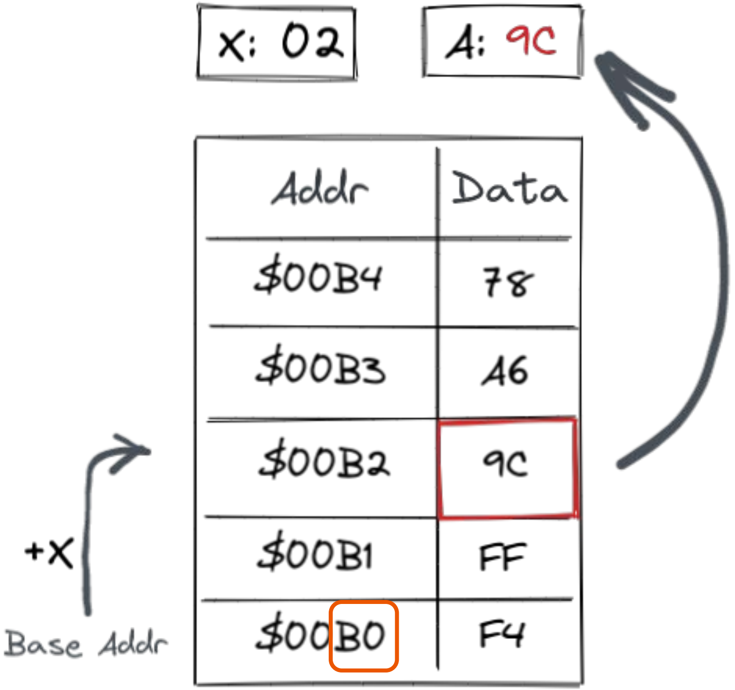
6502 addressing modes (1)

- *Zero page indexed with X or Y* `zp,X/zp,Y`

Example: `LDA $B0,X`

Loads content of address `$00B0+X` `($00B2)` into A.

(Later we'll see we use this to address our FORTH data stack)



Addr	Data
$00B4	78
$00B3	A6
$00B2	9C
$00B1	FF
$00B0	F4

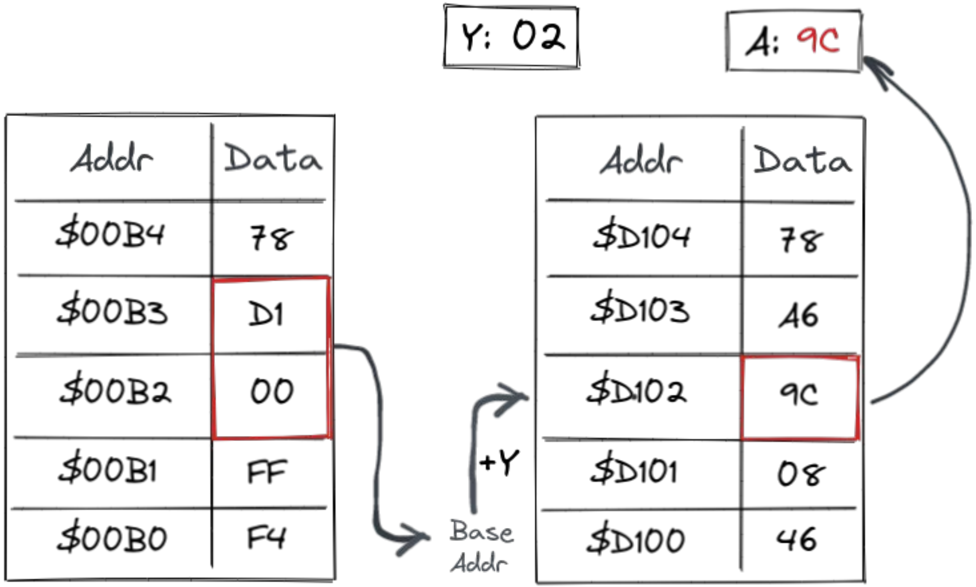
6502 addressing modes (2)

- Zero page **indirect indexed with Y** `(zp),Y`

Example: `LDA ($B2),Y`

Take what's in:
`$00B2` → low byte: `00`
`$00B3` → high byte: `D1`
And forms the base address: `$D100`

Then loads the byte at `$D100+Y` into register **A**.



Addr	Data
$00B4	78
$00B3	D1
$00B2	00
$00B1	FF
$00B0	F4

Addr	Data
$D104	78
$D103	A6
$D102	9C
$D101	08
$D100	46

- Also: Zero page indirect (unindexed) `(zp)`

Implementing FORTH

(My own implementation)

What did I need to implement a minimal FORTH?

- A minimum set of words
  → a dictionary

- A minimal program to be run:
  → a “thread”

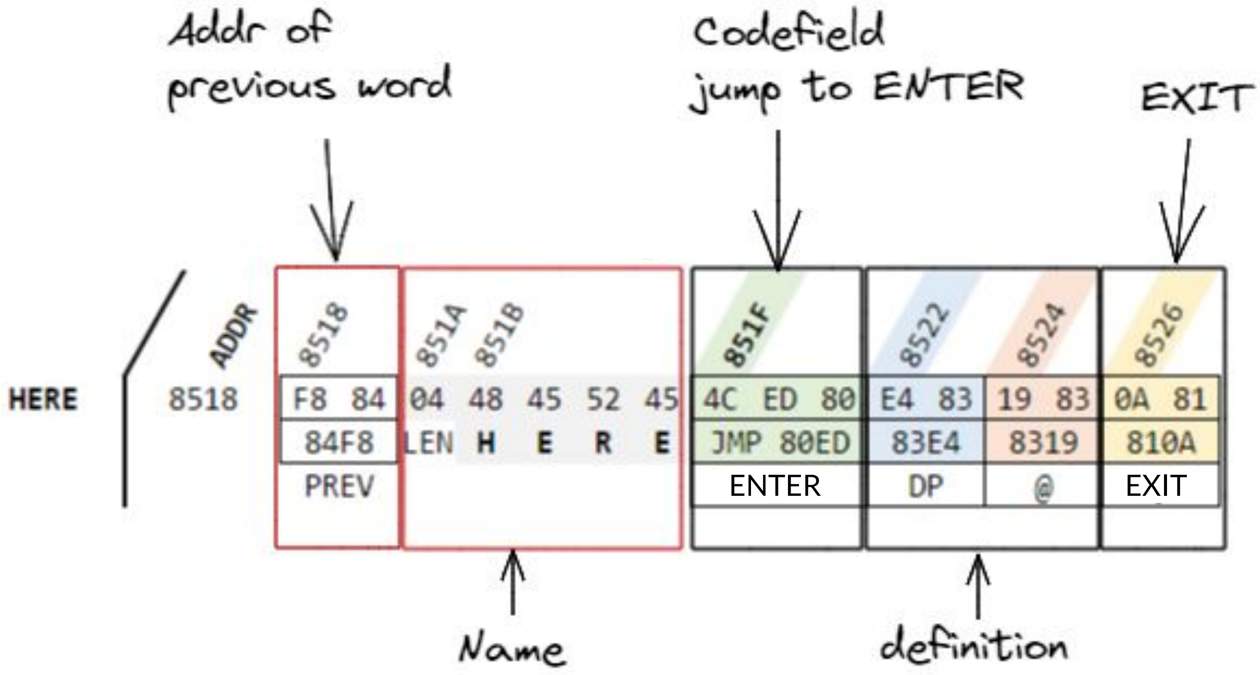
- A way to read and interpret the program, ie. move along the thread:
  → the inner interpreter

- A data stack and a return stack

- And to make it interactive:
  - An outer interpreter
  - A way to lookup words in the dictionary
  - A way to compile new words

- Design choices
  - **DTC**: Direct Threaded Code
  - **16 bits** cells

Dictionary



Addr of previous word

Codefield jump to ENTER

EXIT

ADDR	8518	851A 851B	851F	8522	8524	8526
HERE 8518	F8 84	04 48 45 52 45	4C ED 80	E4 83	19 83	0A 81
	84F8	LEN H E R E	JMP 80ED	83E4	8319	810A
	PREV		ENTER	DP	@	EXIT

Name

definition

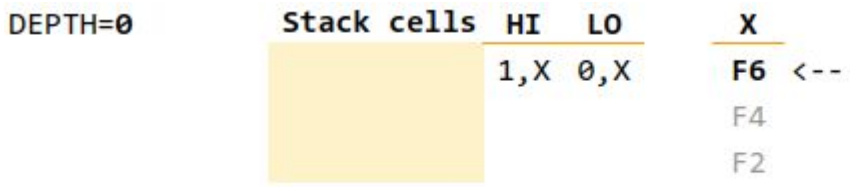
Data Stack

- The Data Stack is build using Zero Page. It starts at the top of ZP (just below the FORTH registers W, IP, G1 and G2), and grows downwards.

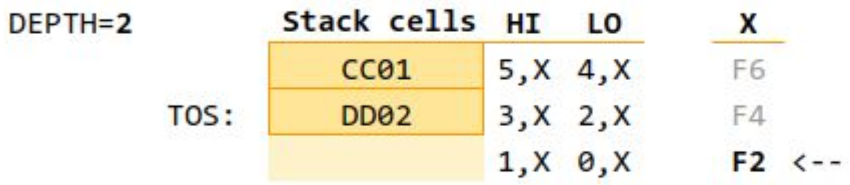
- **Accessing the Data Stack** is easy using the *Zero Page Indexed with X* addressing mode

- `0,X` & `1,X` → next free cell of Data Stack
- `2,X` & `3,X` → cell at Top of the Stack

- Pushing a cell on the stack: storing the Low byte at 0,X, and the high byte at 1,X, and then decrementing X twice (DEX).
- DROP is simply two **`inx`**



```
DEPTH=0        Stack cells   HI   LO        X
                             1,X  0,X       F6 <--
                                            F4
                                            F2
```



```
DEPTH=2        Stack cells   HI   LO        X
                  CC01       5,X  4,X       F6
         TOS:     DD02       3,X  2,X       F4
                             1,X  0,X       F2 <--
```

Inner Interpreter

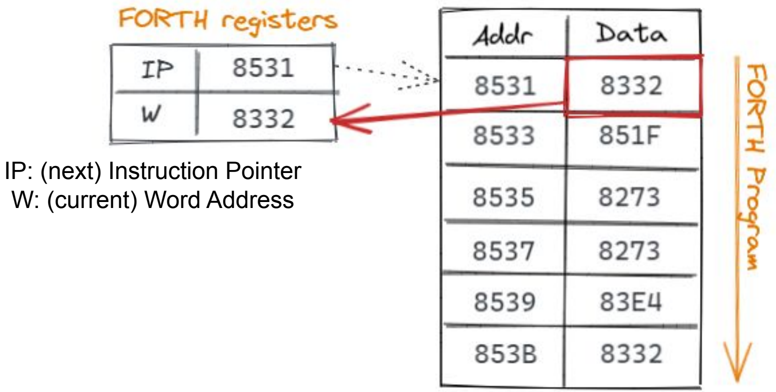
- Two Registers:
  - **IP**: (next) Instruction Pointer
  - **W**: (current) Word Address

- Routines:
  - NEXT
  - ENTER (“COLON” in my implementation)
  - EXIT (“SEMI” in my implementation)

Inner Interpreter - NEXT

- NEXT does 3 things:
  - (IP) --> W
  - IP+2 --> IP
  - JMP (W)

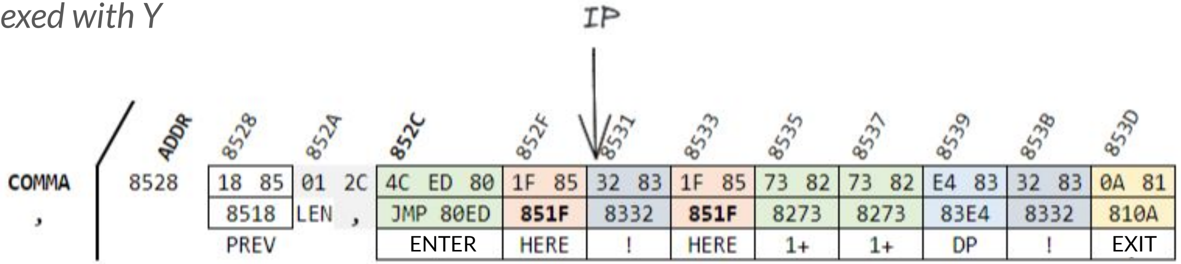
```
NEXT:
; (IP) --> W
        LDA (IP)      ← Zero page indirect (unindexed)
        STA W
        LDY #1
        LDA (IP),y    ← Zero page indirect indexed with Y
        STA W+1
; IP+2 --> IP
        CLC
        LDA IP
        ADC #2
        STA IP
        BCC @skip
        INC IP+1
@skip:
        JMP (W)       ← Absolute indirect
```



IP	8531
W	8332

Addr	Data
8531	8332
8533	851F
8535	8273
8537	8273
8539	83E4
853B	8332

IP: (next) Instruction Pointer
W: (current) Word Address



	ADDR	8528	852A	852C	852F	8531	8533	8535	8537	8539	853B	853D
COMMA ,	8528	18 85	01 2C	4C ED 80	1F 85	32 83	1F 85	73 82	73 82	E4 83	32 83	0A 81
		8518	LEN ,	JMP 80ED	**851F**	8332	**851F**	8273	8273	83E4	8332	810A
		PREV		ENTER	HERE	!	HERE	1+	1+	DP	!	EXIT

Dictionary representation of ","

Inner Interpreter - ENTER

W

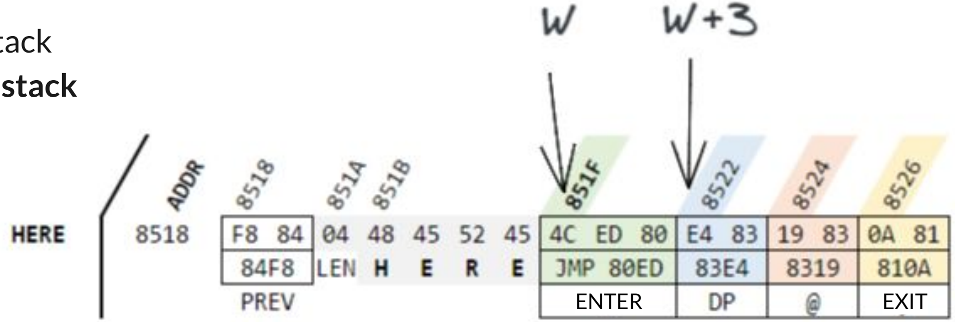
```
defword "ENTER",,
; push IP to Return Stack
      LDA IP+1  ; HI
      PHA
      LDA IP          ; LO
      PHA

; W+3 --> IP
; (Code at W was a JMP)
      CLC
      LDA W
      ADC #3
      STA IP
      LDA W+1
      ADC #0
      STA IP+1
      JMP NEXT
```

← I use the HW stack as **FORTH return stack**

IP ← W+3

Jump to NEXT



	ADDR	8518	851A	851B				851F	8522	8524	8526
HERE	8518	F8 84	04	48	45	52	45	4C ED 80	E4 83	19 83	0A 81
		84F8	LEN	H	E	R	E	JMP 80ED	83E4	8319	810A
		PREV						ENTER	DP	@	EXIT

Inner Interpreter - EXIT

```
defword "EXIT",,
; POP IP from Return Stack
     PLA
     STA IP
     PLA
     STA IP+1
; JMP NEXT
     JMP NEXT
```

← I use the HW stack for FORTH return stack

← Jump to NEXT

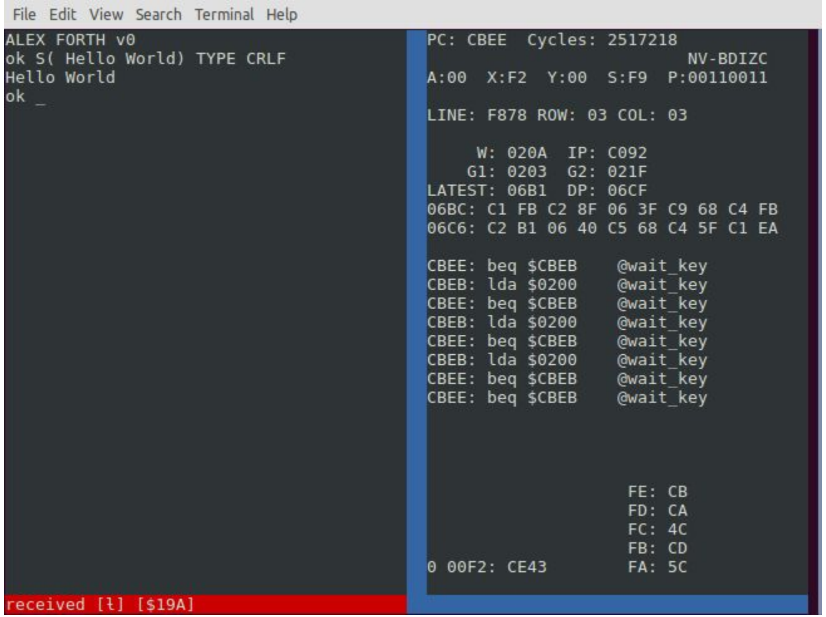
(My) **Development Methodology**

- Iterative: start small and grow incrementally

- First test in emulation:
  - I started with Kowalsky 6502 emulator
  - Then I build my own tools in Python with py65 emulator library

- Then test on hardware:
  - Slow and cumbersome: rom flashing, replace the rom chip
  - Fragile: lots of cables, and breadboard not ideal (I should learn how to do a PCB)

- Commit everything on Github



```
ALEX FORTH v0
ok S( Hello World) TYPE CRLF
Hello World
ok _

received [ł] [$19A]
```

```
PC: CBEE  Cycles: 2517218
                         NV-BDIZC
A:00  X:F2  Y:00  S:F9  P:00110011

LINE: F878 ROW: 03 COL: 03

     W: 020A  IP: C092
    G1: 0203  G2: 021F
LATEST: 06B1  DP: 06CF
06BC: C1 FB C2 8F 06 3F C9 68 C4 FB
06C6: C2 B1 06 40 C5 68 C4 5F C1 EA

CBEE: beq $CBEB    @wait_key
CBEB: lda $0200    @wait_key
CBEE: beq $CBEB    @wait_key
CBEB: lda $0200    @wait_key
CBEE: beq $CBEB    @wait_key
CBEB: lda $0200    @wait_key
CBEE: beq $CBEB    @wait_key
CBEE: beq $CBEB    @wait_key



                    FE: CB
                    FD: CA
                    FC: 4C
                    FB: CD
0 00F2: CE43        FA: 5C
```

My 6502 emulator for developing Alex FORTH for the Cerberus2080

My FORTH Features (for now)

- : ; EXIT
- JUMP EXEC
- IF ELSE THEN
- DO LOOP +LOOP LEAVE
- BEGIN AGAIN, BEGIN UNTIL, BEGIN WHILE REPEAT
- VARIABLE and Local Variables
- MARKER FORGET
- CREATE DOES>
- Comments \ and ( )
- WORDS, HIDE, REVEAL, HIDDEN, RECURSIVE

Limitations (at the moment)

- Doesn’t adhere to any standard
- Only Integer numbers (no floating points)
- Only Hexadecimal representation (no BASE/conversions)
- No stack overflow/underflow verification. Easy to crash!
- Only one dictionary (no contexts)
- Case sensitive words, all predefined words are capital case
- No mass storage, no block feature, no save/restore

Demo time!

Links & how to contact me

My site: https://adumont.github.io/

My FORTH:

- Alex FORTH for 6502 Breadboard Computer
    - Test it in your browser (py65 emulation)

- Alex FORTH for the Cerberus2080 (6502)

Twitter: @adumont

Thank you!