nesdoug

28. ASM part 3

Welcome to part 3 of my 6502 ASM lessons.

Jumping, moves the execution of the program somewhere else.

Examples:

```
  LDA #5
  JMP Skip_Next_Line jump to label 'Skip_Next_Line'
  LDA #7             never does this
Skip_Next_Line:
  STA foo            A = 5, store A at address foo
```

```
Infinite_Loop:
  LDA #5
  JMP Infinite_Loop     jump to the label 'Infinite_Loop'
  STA foo               never does this
```

Indirect jumping

One way to control flow of a program is to have an array of addresses of various parts of the program, and jump to them indirectly.

```
LDA program_state load A from address program_state
ASL A             multiply by 2, since each address is 2 bytes long
TAX               transfer A to X
LDA ADDRESSES, X  load A from ADDRESSES + X (low byte of an address)
STA jump_address  store A at jump_address
LDA ADDRESSES+1, X load A from ADDRESSES+1 + X (high byte of an address)
STA jump_address+1 store A at jump_address+1
JMP (jump_address) jump to the address pointed to by jump_address
```

```
ADDRESSES:
.word FUNCTION1, FUNCTION2
the assembler will replace these with the address of each label



FUNCTION1:
 ...



FUNCTION2:
 ...
```

*Warning, there is a bug related to indirect jumps. The first byte of the indirect jump address can't be on the last byte of a page (such as $3ff). Rather than fetching the second byte from the next page $400, it will fetch the second byte from the same page $300. In the example above, jump_address can't be located $xff. The addresses of Function1 and Function2 can be anywhere.

Sub-Routines

When you use JSR, you jump to the label, and save the return address on the stack. Once that sub-routine is complete, use RTS to return from where we were before. (the program will pull the return address from the stack)

```
LDA #2
JSR Multiply_16
STA foo         A=32, store A at address foo



...
Multiply_16:
  ASL A
  ASL A
  ASL A
  ASL A
  RTS
```

Branching

First lets review the 6502 processor status flags.

c = carry flag
z = zero flag
i = interrupt flag
d = decimal mode (a removed feature, not functioning on the NES)
v = overflow flag
n = negative flag

Most of these flags are useful for comparisons and flow control (branching). Here are some instructions that will set/clear various flags.

CLC = clear the carry flag
SEC = set the carry flag
CLI = clear the interrupt disable flag (allows IRQ interrupts to work)
SEI = set the interrupt disable flag (prevents IRQ interrupts)
CLD = clear decimal flag (set hexadecimal math)
SED = set demical flag (set decimal math) (does not work on the NES)
CLV = clear the overflow flag

It's important to know why and when each flag is set (and which operations won't set flags).

ADC, SBC sets the z, n, c, v flags flags
AND, ORA, EOR sets the z,n flags
ASL, LSR, ROR, ROL sets the z, n, c flags
BIT sets the z, n, v flags
CMP, CPX, CPY sets the z, n, c flags
DEC, DEX, DEY sets the z, n flags
INC, INX, INY sets the z, n flags
LDA, LDX, LDY sets the z, n flags
TAX, TXA, TAY, TYA sets the z, n flags
PLA sets z, n flags
JMP, JSR, RTS, and BRANCHES do not set any flags
STA, STX, STY do not set any flags
PHA and PHP do not set any flags
PLP changes ALL the flags…that's what it's supposed to do

Between the event that set a flag, and the logic that handles the flag, it is safe to store the value somewhere, and safe to branch/jump to another location.

*note: RTI will wipe all your flags, and replace them from a value stored in the stack. When an NMI or IRQ occur, it pushes the processor status and return address to the stack. RTI will return from these interrupts, and it will restore processor status flags (but not the A,X,Y register values).

COMPARISONS (using flags to branch)

CMP compares A to a value
CPX compares X to a value
CPY compares Y to a value

Comparisons work as if a subtraction happened, but without changing the value of A. So think of CMP #5 as SEC,SBC #5. (Also, you don't need to SEC before CMP.)
If the result is zero, z = 1, else z = 0.
If the result is negative, n = 1, else n = 0.
If the A < value, c = 0. If A >= value, c = 1.
OK, now some branching examples.

```
LDA foo
CMP bar   does foo = bar ?
BEQ They_are_equal
 if zero flag set, branch to They_are_equal
BNE They_are_not_equal
 if zero flag not set, branch to They_are_not_equal



They_are_equal:
  ...
  JMP Next_code



They_are_not_equal:
  ...



Next_code:
```

```
LDA foo
CMP #1   does foo = 1 ?
BEQ Foo_Is_One
 if zero flag set, branch to Foo_Is_One
BNE Foo_Is_Not_One
 if zero flag not set, branch to Foo_Is_Not_One
```

```
Foo_Is_One:
  ...
  JMP Next_code



Foo_Is_Not_One:
  ...



Next_code:
```

Also, we can use BEQ/BNE to test if a value is zero, because LDA/LDX/LDY sets the zero flag if the value being loaded is zero.

```
LDA foo     if foo = 0, zero flag set
BEQ Foo_is_zero
BNE Foo_is_not_zero



Foo_is_zero:
  ...
  JMP Next_code



Foo_is_not_zero:
  ...



Next_code:
```

I discourage the use of CMP with BMI and BPL. You should use BCC and BCS for > < comparisons. Here's an example without CMP.

*note $80-ff are considered negative. $0-$7f are considered positive. Look at them in binary…
$80 = 10000000
$7f = 01111111

So, if the upper bit = 1, it's considered negative. If 0, positive.

```
LDA foo   if foo = negative, n flag set
BMI Foo_is_negative  branch if n flag set
BPL Foo_is_positive  branch if n flag not set



Foo_is_negative:
  ...
  JMP Next_code



Foo_is_positive:
  ...



Next_code:
```

Comparisons. BCC is equivalent to 'Branch if Less Than'. BCS is equivalent to 'Branch if Greater Than or Equal'.

(if foo < 40)…branch

LDA foo
CMP #40
BCC Somewhere branch if foo < 40

(if foo <= 40)…branch

LDA foo
CMP #40
BCC Somewhere branch if foo < 40
BEQ Somewhere branch if foo = 40

or…

LDA foo
CMP #41
BCC Somewhere branch if foo < 41

or…reverse them

LDA #40
CMP foo
BCS Somewhere branch if 40 >= foo

(if foo >= 40)…branch

LDA foo
CMP #40
BCS Somewhere branch if foo >= 40

(if foo > 40)…branch

LDA foo
CMP #41
BCS Somewhere branch if foo >= 41

or…reverse them…

lda #40
CMP foo
BCC Somewhere branch if 40 < foo

And, CPX for X register. CPY for Y register. They work the same as CMP…
LDX foo
CPX #41
BCS Somewhere branch if foo >= 41

More about BCC and BCS

There are many, many more uses for BCC and BCS.

Let's say, you want to add numbers, but if result > 255, you want to force it to stay at 255. This works because, if the result of ADC is over 255, the carry flag is set.

```
LDA foo
CLC
ADC #5   ;carry will only be set if result > 255
BCC Still_Under_256 ;branch if carry clear
  LDA #255
Still_Under_256:
```

Similarly, say you want to subtract, but if the result < 0, you want to keep it at zero.

LDA foo
SEC
SBC #5

BCS Still_Zero_Or_More
LDA #0
Still_Zero_Or_More:

You can also use BCC and BCS with ASL/LSR, ROL/ROR. Maybe, you want to use LSR as a modulo 2… to see if a number is even or odd.

```
LDA foo
LSR A   ;bit-shift right, rightmost bit goes into carry flag
BCC Foo_is_Even
BCS Foo_is_Odd



Foo_is_Even:
  ...
  JMP Next_Code
Foo_is_Odd:
  ...
Next_Code:
```

One more kind of comparison…

BIT, test a memory without affecting any register A,X,Y.
8-bits are like…(76543210)
bit 7 goes to n (negative flag)
bit 6 goes to v (overflow flag)

example:
BIT foo
BMI foo_is_negative

Also, you can test specific bits of a RAM address, as if an AND instruction were used. LDA a value… let's say #1. BIT $30 (tests the bit zero of RAM address 0x30). If it's 1, Z (zero flag) = 0. If it's 0, Z = 1. That is…if the result of an AND of the 2 values is zero or not, it affects the Z flag.

Another example. $30 is #2. The A register is #2. BIT $30 will clear the z flag. You would use BEQ (branch if z set) or BNE (branch if z not set) to control flow from here.
*final note:you can only branch +127 or -128 bytes (relative to the byte after the branch instruction. Any more and the assembler will give you branch-out-of-range errors. The standard solution is to replace those long branches with the opposite branch, and a jump to the label.

BEQ label
…over 127 bytes of code…error, too far
label:
…….replace it with…

```
BNE :+
JMP label
:
…
label:
```

: is an unnamed label. :+ means branch forward to the next unnamed label. :- means branch backwards to the next unnamed label.

March 13, 2016August 6, 2017 dougfraker