nesdoug

27. ASM part 2

Bit Shifting

The 6502 has 2 ways of shifting bits left and right. In these examples, I will number each bit…there are 8 bits, numbered 0-7. Only the accumulator (A register) can do bit shifts and bitwise operations. Also, you can do bit shifting to a RAM address, without affecting A.

LSR – shift right

zero -> 76543210 -> carry flag

ROR – roll right

old carry flag -> 76543210 -> new carry flag

ASL – shift left

carry flag <- 76543210 <- zero

ROL – roll left

new carry flag <- 76543210 <- old carry flag

LSR shifts all the bits right one position, and a zero goes into the highest bit. Effectively, this is the same as dividing by 2, with some rounding error.

```
zero -> 00010000
        00001000, carry = 0
```

```
zero -> 00001111
        00000111, carry = 1
```

ROR works the same as LSR, expect the old carry flag goes in rather than zero.

ASL shifts all the bits left one position, and a zero goes into the lowest bit. Effectively, this is the same as multiplying by 2. Right to left here…

```
   00010000 <- zero
<- 00100000
carry = 0
```

```
   11110000 <- zero
<- 11100000
carry = 1
```

ROL works the same as ASL, expect the old carry flag goes in rather than zero.

Now, some examples, using C programming examples to start with.

foo = bar << 2; // 8-bit numbers

```
LDA bar  load A from address bar
ASL A    bit-shift A left
ASL A    bit-shift A left
STA foo  store A to address foo
```

```
foo = bar >> 3; //8-bit numbers
```

```
LDA bar  load A from address bar
LSR A  bit-shift A right
LSR A  bit-shift A right
LSR A  bit-shift A right
STA foo  store A to address foo
```

```
And, here's some 16-bit examples.
```

foo = bar << 2; // 16-bit numbers

```
LDA bar+1  load A from address bar+1 (the high byte)
STA foo+1  store A to a address foo+1 (the high byte)
LDA bar    lda A from address bar (the low byte)
ASL A      bit-shift A left (high bit shifted into carry flag)
ROL foo+1  bit-shift left address 'foo+1', rolling that carry flag in
ASL A      bit-shift A left (high bit shifted into carry flag)
ROL foo+1  bit-shift left address 'foo+1', rolling that carry flag in
STA foo    store A to the address foo (the low byte)
```

foo = bar >> 3; //16-bit numbers

```
LDA bar   load A from address bar (the low byte)
STA foo   store A to the address foo (the low byte)
LDA bar+1 load A from the address bar+1 (the high byte)
LSR A     bit-shift A right (low bit shifted into carry flag)
ROR foo   bit shift right address 'foo', rolling that carry flag in
LSR A     ...
ROR foo
LSR A
ROR foo   ...3 times
STA foo+1 store A to the address foo+1 (the high byte).
```

Bitwise Operations.

AND, OR, and XOR…called here AND, ORA, and EOR. These things only work with the A register.

Here's what AND does. bit by bit.

AND #value
A, AND value= result

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

AND only sets a bit if both bits in A and value are 1.

Example:

```
A      00010001
value  00000101
result 00000001
```

AND is used to isolate a single bit. The way I handle button presses, I roll them into joypad1. If I want to find out if the Left button is being pressed…I know that it is this bit of joypad1 00000010 ($02). Here's how the code would usually go…

LEFT = $02 ;defined at the top of the page

```
LDA joypad1  load A from address joypad1
AND #LEFT    AND A with value 2, result now in A
BEQ :+       branch if the result is zero (to unnamed label)
             skipping this next line of code
JSR Left_Pressed jump to sub-routine handline left button presses
:            just a label
```

Another use for AND, is to 'mask' out certain bits. Let's say, I have a peice of data, where the upper bit is a special flag, and the lower 7 bits is the data. If I want just the data, I would AND #$7f (01111111) to remove the upper-bit…

```
LDA data
AND #$7f
```

ORA (bitwise OR operation)

Here's what ORA does. bit by bit.

```
ORA #value
A, ORA value= result
0 ORA 0 = 0
0 ORA 1 = 1
1 ORA 0 = 1
1 ORA 1 = 1
```

If either A or the value has a bit set, the result will have that bit set.

```
Example:
A      00010001
value  00000101
result 00010101
```

ORA is a way ensure that certain bits are set, without effecting the other bits (as math would do).

Music code is a good example. The left 4 bits control the sound. The right 4 bits volume. So, if you want to keep the 'instrument' the same, the first 4 bits would always be $C (for example), while the volume may change. So, you might store $c0 in variable 'instrument' and store the volume in variable 'volume'. When you need to combine them, you would use ORA.

```
LDA instrument  instrument is $c0
ORA volume      volume is 0 - $0f
STA $4000       result stored to music register, address $4000.
```

EOR (exclusive OR operation), means one or the other, but not both.

```
EOR #value
A, EOR value= result
0 EOR 0 = 0
0 EOR 1 = 1
1 EOR 0 = 1
1 EOR 1 = 0
```

```
Example:
A      00010001
value  00000101
result 00010100
```

EOR is usually used to get the negative value of a number. Say you have -5 ($fb) and you want to turn it into 5, you EOR #$ff and add 1. The same for converting back to -5.

```
LDA foo   Let's say foo = $fb (-5)
EOR #$ff  A = 4 now
CLC
ADC #1    A = 5 now
```

and reverse works too…

```
LDA foo   Let's say foo = 5
EOR #$ff  A = $fa now
CLC
ADC #1    A = $fb now (-5)
```

TRANSFERRING registers

TAX A transfers to X
TXA X transfers to A
TAY A transfers to Y
TYA Y transfers to A

TXS X transferred to the stack pointer
TSX stack pointer transferred to X

This is the only way to access the stack pointer. Usually, the stack pointer is set to $ff at the start of the program and never thought of again.

```
LDX #$ff load X with value $ff
TXS      transfer to stack pointer
```

(the stack grows down from $1ff)

More Stack Operations
PHA push A to the stack (and adjust the stack pointer -1)
PLA pull A (pop A) from the stack (and adjust the stack pointer +1)

PHP push the processor status to the stack (and stack pointer -1)
PLP pull the processor status from the stack (and stack pointer +1)

Unfortunately, you can't push a few arguments to the stack, jump to a sub-routine and then use those numbers…not easily, at least. Because, the jump to sub-routine also pushes the return address to the stack, on top of your numbers. PHA and PLA can be used as a cheap local variable. But, be careful. If

you're inside a sub-routine and you PHA, and forget to PLA, your program will crash when it tries to pull the return address, and gets your PHA number instead.

A few more things…

NOP does nothing but wastes 2 cycles of CPU time
BRK a non-maskable interrupt…will jump the program to wherever the BRK vector tells it…this usually only happens if a big error has occured, as the machine code for BRK is #00…which indicates that the program has branched to an area of the ROM with nothing there.

And, next time we will go over jumping, branching, and comparison.

March 13, 2016April 13, 2017 dougfraker