Machine Language Tutorial Part 3 - Program Flow

Before We Start...

Before we proceed any further we must talk about numbers in machine language. Most monitors allow for denoting numbers in multiple forms- the monitor converts it to hex for you.

To enter a hexadecimal number into the monitor, put a dollar sign before the number.

```
LDA #$20
STA $20
LDA $2000
```

Most more modern monitors (like the built-in VICE one) also support binary numbers. Use a percent sign.

```
LDA #%00011011
AND #%11101001
```

Monitors typically don't support this, but assemblers do: Don't use any sign at all for decimal numbers.

```
LDA #153
STA 56
STA 53280
```

Signed and Unsigned Numbers

Signed numbers are numbers that can have a positive or negative value. To represent this we use the leftmost bit (MSB, Most Significant Bit).

```
#%10010010 - Negative
#%01101101 - Positive
```

Thus we get a range from -128 ($80) to 127 ($7F) for a one-byte value. The computer doesn't care if a number is signed or not: you must write your program to deal with them.

If a value is unsigned, then it cannot have a sign and is always positive.

If you were to subtract 1 from unsigned $00, you would just roll over to $FF. But for a signed $00, you would also get $FF- but this time it represents -1. Decreasing of a signed number goes like this:

```
$02 (2)
$01 (1)
```

```
$00 (0)
$FF (-1)
$FE (-2)
and then the maximum for one byte would be:
$80 (-128)
Subtracting any further would make:
$7F (127)
This would be overflow, since our value is only one byte.
```

Jumps & Branches

To make an unconditional jump to another memory address, we use JMP.

```
JMP $1234
```

It does not matter what the conditions are: if JMP is encountered then the program counter will always change to the address mentioned.

Here's something easy you can do with jumps:

```
A 1000 INC $D020
JMP $1000
```

What does this do? It increases the number of the border color (at $D020), and then jumps back. It does this so fast that it changes the color while the screen is being drawn, so you get a lot of lines.

Branches are conditional: there are several conditions possible for a branch. Right now the only ones we care about are:

```
BNE - Branch if not equal
BEQ - Branch if equal
```

But how do we make the computer see if the value is equal? We use compare instructions.

```
CMP - Compare A with
CPX - Compare X with
CPY - Compare Y with
```

So to check if X was the same as $1234, and then branch to $1000 if not equal to, we'd use:

```
CPX $1234
BNE $1000
```

If the branch was not taken then execution continues.

One odd thing about branches is that they are made of only two bytes. This is because the second byte is a signed number that tells the computer how far to branch: so therefore you can only branch back 128 bytes and forward 127. This is called relative addressing.

So instead of doing this:

```
A 1000 CPX #$50
BNE $2000
TXA
* more than 127 bytes of code *
A 2000 TYA
```

You'll need to get around this with an unconditional jump, like so:

```
A 1000 CPX #$50
BEQ $1007
JMP $2000
TXA <- this is $1007
* more than 127 bytes of code *
A 2000 TYA
```

Subroutines

Subroutines are freely callable routines that can reduce memory use for often-used functions. To jump to a subroutine we use JSR. The subroutine goes until it encounters an RTS (return from subroutine). So we might have a subroutine we want to jump to at $1234:

```
JSR $1234
LDA #$10
```

And this might be at $1234:

```
STA $4321
RTS
```

So, we call the subroutine and then STA $4321 is executed. It encounters an RTS, so it goes back and executes LDA #$10.

If an RTS is encountered without being in any of your own subroutines, the computer will go back to BASIC (will crash if you got the ROMs off). There are several Kernal routines, but these are not suitable for demos- only use them for tools and things that don't need to be fast.

Flags

The flags are several bits in the status register (NV-BDIZC) that tell the CPU things about the past operation. (e.g. the BNE from earlier.)

Zero Flag (Z)

This flag is set (1) if the result of a compare instruction was equal or the result of a previous

instruction was zero, else it's clear (0). So if we did this:

```
LDX #$01
DEX
```

then DEX would make X zero, setting the zero flag. It will also be set if a load or store instruction resulted in 0.

The BNE instruction branches if the zero flag is clear, and BEQ branches if set. So we could do a simple loop with X like this:

```
A 1000 LDX #$08
* your code *
DEX <- this is $1002 in this case
BNE $1002
```

So this doesn't do much, it just loops DEX 8 times.

Carry Flag (C)

The carry flag is set if the result of a compare instruction was greater than or equal to, else it's clear. It also acts as a carry and borrow in addition and subtraction, and is important in multiplication and division.

BCC (branch if carry clear) branches if clear and BCS branches if set. CLC clears carry and SEC sets carry.

Negative Flag (N)

The negative flag is set/clear depending on what the MSB was after an operation. So if the MSB was 1, the flag would be set. So LDA #$A0 would set the N flag, because $A0 has the MSB as 1.

BPL (branch if plus) branches if clear and BMI (branch if minus) branches if set.

Overflow Flag (V)

The overflow flag is usable pretty much only for signed numbers. It is set if the result changed the sign from negative to positive, or positive to negative in the wrong way. It only applies to math, not incrementing or decrementing. So for example, if $7F had $1 added, it would have a result of $80: which is 127 to -128, so that's overflow and would set the overflow flag.

BVC branches if clear and BVS branches if set. CLV clears overflow.

End of part 3.