Round 2 • Lookup Table (LUT)

OCA / Atlantis

// Introduction

In this article, we continue exploring text mode, and examine a few solutions to the previous challenge.

// draw_char challenge implementations

In the previous Round we left off with a challenge about writing a routine that could draw a char anywhere on the screen.

```
BasicUpstart2(main)
main:
    lda #'a'  // or lda #1
    sta char
    lda #10
    sta row
    lda #12
    sta col
    jsr draw_char

forever:
    jmp forever

draw_char:
   // The challenge is to write
   // this routine.
   rts

row:  .byte 0
col:  .byte 0
char: .byte 0
```

Listing 1: 01_challenge.asm

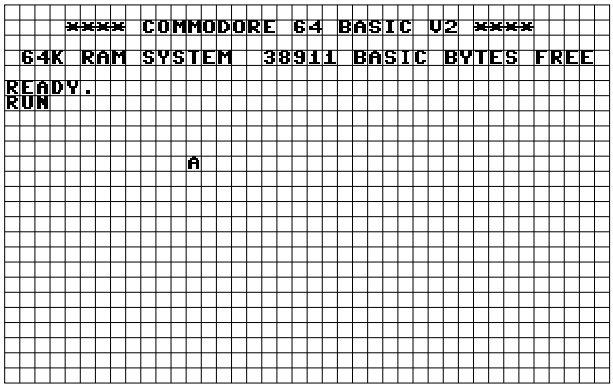


Figure 1: expected output for row=10 and col=12

To find the address of the char, we need to compute this formula:

$$\$0400 + \text{ROW} \times 40 + \text{COL}$$

The "+ COL" part should be no problem at all, as there is an opcode to perform addition, but there is no opcode at all on the C64 to perform a multiplication. So we have to find a way to multiply by 40.

// Performing a multiplication as a loop of addition

$$\text{Well...}\quad 3 \times 40 = 3 + 3 + 3 + (\text{well... 37 more})$$

So let's give this idea a try. We could start by looping "row" times

```
  ...
  ldx row
  beq end   // Can you guess why we need this?
loop:
    ...
    dex
    bne  loop
end:
  ...
```

and then do the 16-bit addition of "+40" inside this loop. The whole code should look like this.

```
.const WIDTH  = 40
.const HEIGHT = 25
.const SCREEN_ADR = $0400

// ZP
.pc =$02 "== ZeroPage (ZP) ==" virtual
    screen_ptr: .word 0

BasicUpstart2(main)
main:
    lda #'a'  // or lda #1
    sta char
    lda #10
    sta row
    lda #12
    sta col
    jsr draw_char

    jmp *
```

```
draw_char:
    lda #>SCREEN_ADR
    sta screen_ptr+1
    lda #<SCREEN_ADR
    sta screen_ptr

    ldx row
    beq end

// ROW * WIDTH
loop:
    clc
    adc #WIDTH
    bcc !+
        inc screen_ptr+1
    !:
    dex
    bne  loop

    sta screen_ptr

end:
    // Use Y for "+ COL"
    ldy col
    lda char
    sta (screen_ptr),y
    rts

.pc = * "== Data =="
// We could also put these in ZP
row:  .byte 0
col:  .byte 0
char: .byte 0
```

Listing 2: 02_draw-char-loop.asm

> **Side note:** If you wonder what !+ and !: are, there is an explanation a bit later in this document about labels.

Let's compile this code using the "showmem" argument to get the Memory Map

```
java -jar KickAss.jar -showmem 02_draw-char-loop.asm

...
...
Memory Map
----------
Default-segment:
  *$0002-$0003 == ZeroPage ==
  $0801-$080c Basic
  $080e-$0844 Basic End
  $0845-$0847 == Data ==
```

I guess this is a good habit to check where your data are located, and to name your code/data sections.

The lines starting with a * are the virtual blocks, i.e. the ones that are not going to be included in your prg.

Let's now try another way, by doing a proper multiplication by 40...

// Multiplying by 40

As we already saw, there is no opcode for multiplication. But we can shift the bits... and shifting the bits to the left is actually a multiplication by a factor of 2...

Let's try with 3

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	0	1	1

and shift the bits to the left, this gives us 1 1 0, which is 6 in decimal, so we just did a multiplication by 2

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	1	1	0

shift again? 1 1 0 0 is 12 in decimal.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	1	1	0	0

But what happens if bit7 is set to 1 prior to shifting? Let's say we start from 129

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	0	0	0	0	0	0	1

and do the left shift

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	0	1	0

oops... but the 1 we had in bit7 is not lost, it has been transferred to the Carry flag. (i.e. that's something we can now test using bcc or bcs)

The opcode to use to shift left is **ASL**... and to shift right, so as to divide by 2, this is **LSR**

But this is not all, we just saw we can shift the bits, but we can also rotate them with **ROL** (rotate left) and **ROR** (rotate right)

Rotating is very similar to shifting, but not exactly the same, as we can see in Figure 2

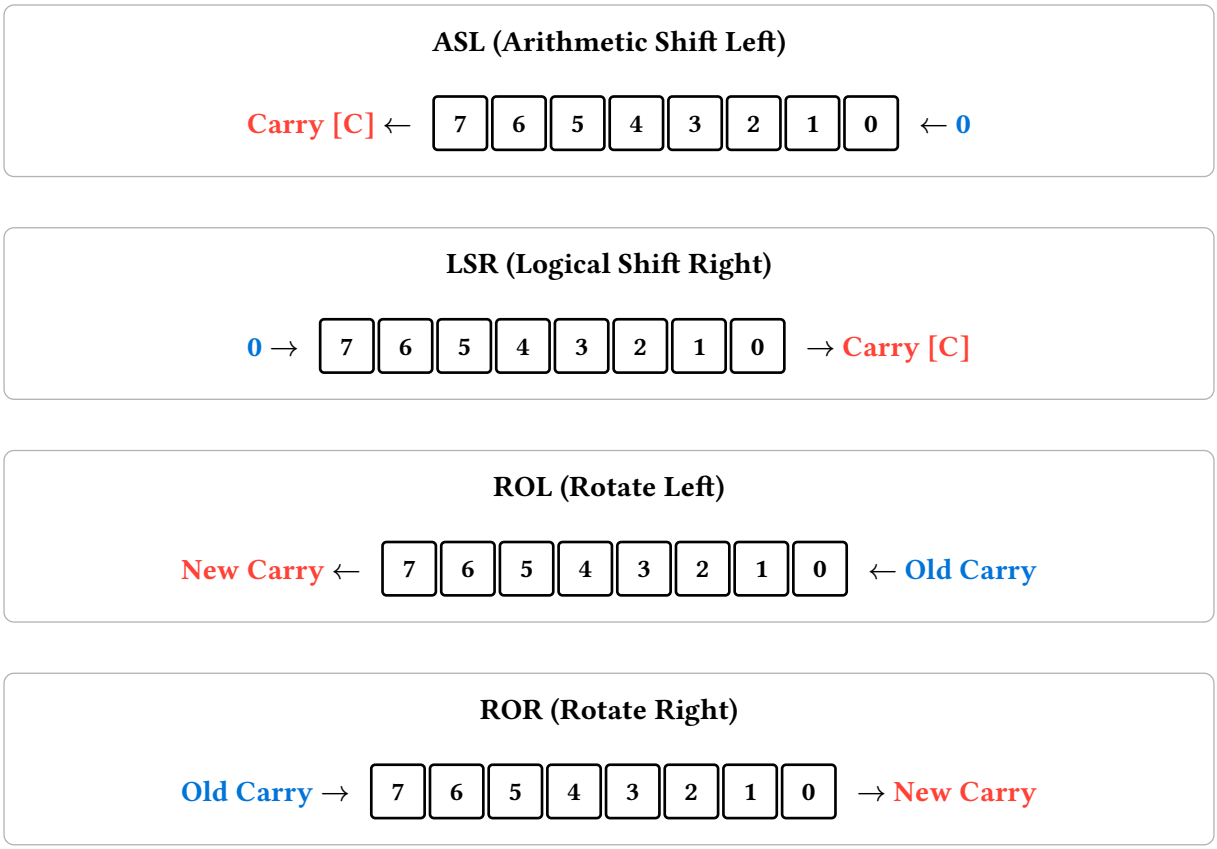


Figure 2: ASL, LSR, ROL and ROR

> **Side note:** Nice, but what is Carry in the first place?
>
> Carry (or 'C') is one of the CPU status flags. A flag is a single bit—either '0' or '1'—stored in the CPU's Processor Status register. The Carry flag is set (1) or cleared (0) depending on arithmetic and shift operations.

)

OK, that means that using left shifting and rotation, and managing Carry properly to handle 16-bit values, we could multiply by 2, 4, 8, 16, 32, 64... **but not by 40...**

So here is the trick:

$$\text{NUMBER} \times 40 = \text{NUMBER} \times 32 + \text{NUMBER} \times 8$$

and both 8 and 32 are multiplications we can do with shifting, so we are all set... here is the whole code

```
.const SCREEN_ADR = $0400

.pc =$02 "ZP" virtual
row:  .byte 0
col:  .byte 0
char: .byte 0
screen_ptr: .word 0

BasicUpstart2(main)
main:
    lda #'a'
    sta char
    lda #10
    sta row
    lda #12
    sta col
    jsr draw_char

    jmp *
```

```
draw_char:
    // Calculate A = row * 8
    lda row
    asl
    asl
    asl
    sta screen_ptr // Low byte of (row * 8)
    lda #0
    rol
    sta screen_ptr+1 // High byte of (row * 8)

    // Save (row * 8) low byte in Y for the
    // 16-bit addition
    ldy screen_ptr

    // Shift screen_ptr left twice
    // to get row * 32
    asl screen_ptr
    rol screen_ptr+1
    asl screen_ptr
    rol screen_ptr+1

    // Add (row * 8) to (row * 32)
    // -> gives row * 40
    tya
    clc
    adc screen_ptr
    sta screen_ptr
    lda screen_ptr+1
    adc #0  // this isn't really "+0"...
            // what is it actually adding,
            // and why?
    sta screen_ptr+1

    // Add base screen address ($0400)
    lda screen_ptr+1
    clc
    adc #>SCREEN_ADR
    sta screen_ptr+1

end:
    // Use Y for "+ COL"
    ldy col
    lda char
    sta (screen_ptr),y
    rts
```

Listing 3: 03_draw-char-mult40.asm

Let's compile this code using the "showmem" argument to get the Memory Map

```
java -jar KickAss.jar -showmem 03_draw-char-mult40.asm

...
...
Memory Map
----------
Default-segment:
  *$0002-$0006 ZP
  $0801-$080c Basic
  $080e-$084f Basic End
```

The good news is that there is another way, faster, and simpler, but taking a bit more memory. Given there is no opcode for doing multiplication, what if we completely avoid doing that computation? Given that we "only" have 25 rows, we can pre-compute all the multiplications by 40 for each of the 25 rows, and store all this into a table in memory. Such a table is called a lookup table.

// Lookup Tables (LUT)

Defining a lookup table in Kick Assembler is very easy. Let's say we would like to keep 10 values (0 to 9) for a multiplication factor of 5.

```
0 ->  0
1 ->  5
2 -> 10
3 -> 15
...
9 -> 45
```

```
mult_by_5_lut:
  .for (var i = 0; i< 10; i++){
    .byte i * 5
  }
```

> **Side note:** Wait... a for loop in assembly??? Well, this is not assembly, this is a Kick Assembler directive. Directives are executed when you build your prg (so at compilation time, not at runtime). We will talk more about directives and macros in a later article.

Here is a shorter form

```
mult_by_5_lut:
 .fill 10, i * 5 // when using .fill, there is a variable i we can use that
represents the current iteration
```

mult_by_5_lut is going to end up with an address somewhere in memory. Let's say we force $1000 as an example.

```
.pc = $1000
mult_by_5_lut:
 .fill 10, i * 5
```

The memory starting at $1000 will look like this

$1000	0
$1001	5
$1002	10
$1003	15
$1004	20
$1005	25
$1006	30
$1007	35
$1008	40
$1009	45

But how to use that lookup table? Let's make a quick testing prg...

```
BasicUpstart2(main)
main:

  lda $1003   // A will be 15
  // put the corresponding char
  // (15 -> 'O' ) in the
  // top/left corner
  sta $0400

  lda mult_by_5_lut + 3 // same A will be 15
  sta $0402

  ldx #3
  lda mult_by_5_lut, x // same A will be 15
  sta $0404

  jmp *

.pc = $1000
mult_by_5_lut:
.fill 10, i * 5
```

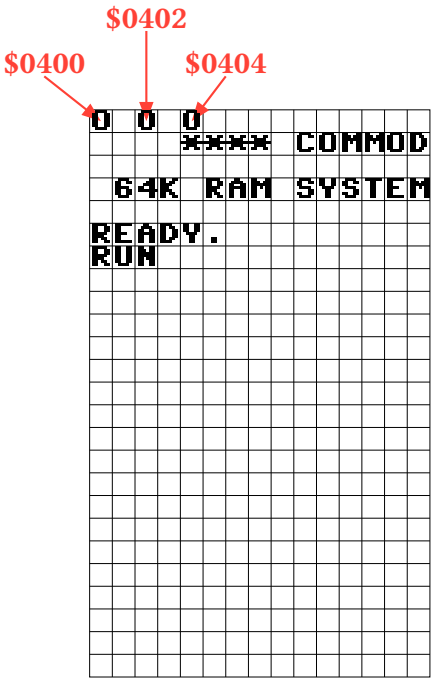


Figure 3: expected output

> **Side note:** A will be 15? What does that mean?
>
> 'A' (or 'Acc') stands for the **Accumulator**. It is the main CPU register used to hold values you want to manipulate or work with.
>
> - `lda` is the mnemonic for **Load Accumulator** (puts a value into 'A').
> - `sta` is the mnemonic for **Store Accumulator** (writes the value inside 'A' into a memory location).

Good, we can now use lookup tables to precompute values, but we have to multiply by 40, not by 5... and the maximum value is going to be 40 x 24 = 960 (24 and not 25 because the rows go from 0 to 24).

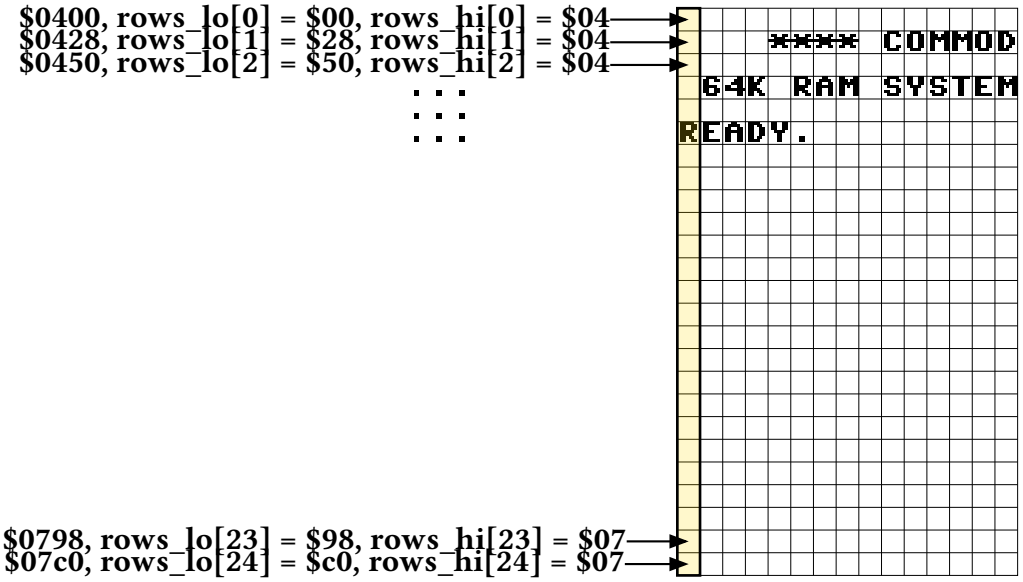
960 is not going to be a number we can store in a single byte (0-255).

So we need two lookup tables, one to hold the lower 8 bits of the value and one to hold the higher 8 bits of the value.

```
  mult_by_40_lo:
    .fill 25, <i*40
  mult_by_40_hi:
    .fill 25, >i*40
```

Good… but we can go even further: we could store the address of the first char of each single row, by including SCREEN_ADR directly into the lookup table.

```
rows_lo:
    .fill HEIGHT, <SCREEN_ADR + i*WIDTH
rows_hi:
    .fill HEIGHT, >SCREEN_ADR + i*WIDTH
```



Now let's have a look at the whole code using lookup tables

```
.const WIDTH  = 40
.const HEIGHT = 25
.const SCREEN_ADR =
$0400

.pc =$02 "ZP" virtual
    row:    .byte 0
    col:    .byte 0
    char:   .byte 0
    screen_ptr: .word 0

BasicUpstart2(main)
main:
    lda #'a'
    sta char
    lda #10
    sta row
    lda #12
    sta col
    jsr draw_char

    jmp *
```

```
draw_char:
    ldx row
    lda rows_lo,x
    sta screen_ptr
    lda rows_hi,x
    sta screen_ptr+1

    ldy col
    lda char
    sta (screen_ptr),y
    rts

.pc = * "LUT"
rows_lo:
    .fill HEIGHT, <SCREEN_ADR + i*WIDTH
rows_hi:
    .fill HEIGHT, >SCREEN_ADR + i*WIDTH
```

Listing 4: 04_draw-char-lut.asm

Way easier (and way faster too), but takes a bit more memory as we have to hold the two lookup tables. But these are well-invested bytes, I would say…

Let's compile this code using the "showmem" argument to get the Memory Map

```
java -jar KickAss.jar -showmem 04_draw-char-lut.asm

...
...
Memory Map
----------
Default-segment:
  *$0002-$0006 ZP
  $0801-$080c Basic
  $080e-$082e Basic End
  $082f-$0864 LUT
```

> **Side note:** Doing LUTs for 16-bit values is such a common thing that Kick Assembler provides something for this. So instead of
>
> ```
> rows_lo:
>     .fill HEIGHT, <SCREEN_ADR + i*WIDTH
> rows_hi:
>     .fill HEIGHT, >SCREEN_ADR + i*WIDTH
> ```
>
> You could use .lohifill like this
>
> ```
> rows_lut: .lohifill HEIGHT, SCREEN_ADR + i*WIDTH
> ```
>
> and then use it like this

```
lda rows_lut.lo,x
sta screen_ptr
lda rows_lut.hi,x
sta screen_ptr+1
```

Whole code provided as an alternative in 04_draw-char-lut-alt.asm

// A bit of theory: Labels in Kick Assembler

let's say we would like to do a loop

```
ldx #0
loop:

  // do something here

  inx
  cpx #10
  bne loop
```

Cool, that shoud compile. but now let's say we have two loops.

```
ldx #0
loop:
  // do something here
  inx
  cpx #10
  bne loop

ldx #0
loop:
  // do something else here
  inx
  cpx #8
  bne loop
```

That's not going to compile... we cannot have the label "loop" twice... so what do we do?

Solution 1: use loop1: and loop2:

```
ldx #0
loop1:
  // do something here
  inx
  cpx #10
  bne loop1

ldx #0
loop2:
  // do something else here
  inx
  cpx #8
  bne loop2
```

Yes, that would do. but that's not the only way, we can also prefix a label with a '!' like this

```
ldx #0
!loop:
  // do something here
  inx
  cpx #10
  bne !loop-  // '-' means jump to the previous !loop label

ldx #0
!loop:
  // do something else here
  inx
  cpx #8
  bne !loop- // '-' means jump to the previous !loop label
```

in the same way you can jump to the next ! label with +

```


jmp !skip+

// some code here

!skip:
```

you can have multiple - and +

```


jmp !skip++

// some code here

!skip:
// some code here

!skip:  // jmp !skip++ will jmp here
// some code here
```

And finally you can omit a label name and use just '!:', '!+' and '!-' like in this idomatic 16-bit pointer addition here

```
.pc = $02 "ZeroPage" virtual
ptr .word 0

// some code here to init the ptr

inc ptr
bne !+
inc ptr+1
!:
```

Is that all? no, you can also scope your labels using {}

```
subroutine1: {
  ldx #0
  loop:
    // do something here
    inx
    cpx #10
    bne loop
    rts
}

subroutine2: {
  ldx #0
  loop:
    // do something here
    inx
    cpx #10
    bne loop
    rts
}
```

This will compile fine even if 'loop:' appear twice in this code.

// Next Challenge

All 3 implementations we saw so far are using indirect addressing **sta (pointer),y** Another way is to use Self-Modifying Code (SMC), so literally code that modifies itself.

Could you write an implementation of draw_char using SMC instead of indirect addressing?

// Recap of the Mnemonics we used so far

Round 1

Mnemonic	Description
`lda`	Load Accumulator — Loads a value into register A.
`ldx`	Load X Register — Loads a value into register X.
`ldy`	Load Y Register — Loads a value into register Y.
`sta`	Store Accumulator — Writes the value from register A into memory.
`jmp`	Jump — Sets the Program Counter (PC) to jump directly to a target address.
`jsr`	Jump to Subroutine — Pushes the return address onto the stack and jumps to a subroutine.
`rts`	Return from Subroutine — Pulls the return address from the stack and resumes execution after the `jsr`.
`clc`	Clear Carry Flag — Clears the Carry flag ($C = 0$), required before 8-bit or 16-bit addition (`adc`).
`adc`	Add with Carry — Adds a value and the Carry flag to register A ($A = A + M + C$).
bcc	Branch if Carry Clear — Jumps to a target address if the Carry flag is clear ($C = 0$), e.g. when an addition produced no carry out of the byte (the result stayed ≤ 255).

`bne`	Branch if Not Equal — Jumps to a relative target address if the Zero flag is clear ($Z = 0$).
`dex`	Decrement X Register — Decreases register X by 1 ($X = X - 1$).
`dey`	Decrement Y Register — Decreases register Y by 1 ($Y = Y - 1$).
`inc`	Increment Memory — Increases a value in memory by 1 ($M = M + 1$).

New Mnemonics introduced in Round 2

Mnemonic	Description
`beq`	Branch if Equal (Zero flag = 1). Branches to a relative target address if the result of the previous operation was zero or if a comparison matched.
`asl`	Arithmetic Shift Left. Shifts all bits in the accumulator or memory left by 1. Bit 7 is shifted into the Carry flag (C), and Bit 0 is filled with 0.
`lsr`	Logical Shift Right. Shifts all bits in the accumulator or memory right by 1. Bit 0 is shifted into the Carry flag (C), and Bit 7 is filled with 0.
`rol`	Rotate Left. Rotates all bits in the accumulator or memory left by 1 through the Carry flag. Bit 7 moves into Carry, and the old Carry value moves into Bit 0.
`ror`	Rotate Right. Rotates all bits in the accumulator or memory right by 1 through the Carry flag. Bit 0 moves into Carry, and the old Carry value moves into Bit 7.

// Stuck?

Stuck somewhere in the challenge? Feel free to send me a PM (OCA/Atlantis) on CSDb. I'll be happy to help when I can.

// What's coming next

- See how we could implement draw_char using Self-Modifying Code.