Round 4 • Precomputed Random Numbers

OCA / Atlantis

// Introduction

Last time we left off with the challenge of filling the screen randomly. Let's try a first idea using a precomputed lookup table (LUT) to hold random numbers.

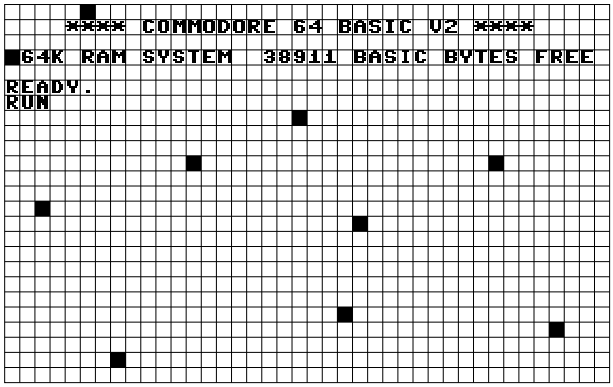


Figure 1: An intermediate position after a few iterations

// Precomputed LUT of random numbers

Kick Assembler provides a random() function. We could use that to fill a lookup table with 1000 precomputed random screen positions.

```
random_lut:
  .fill 1000, floor(random()*1000) // random() returns a number between 0 and 1
```

Before building the whole thing, let's check what these random numbers look like. We need a few changes in order to print them at compilation time.

```
.const NB_RAND=5
.var vals = List()
.for (var i = 0; i < NB_RAND; i++) {
    .eval vals.add(floor(random() * NB_RAND))   // compute once, store in a list
}

random_lut:
    .fill NB_RAND, vals.get(i)                  // fill from the list

.for (var i = 0; i < NB_RAND; i++) {
    .print "index " + i + " = " + vals.get(i)   // print the values
}
```

Here is the output we get when we compile the code:

```
java -jar KickAss.jar -showmem 01_precomputed-random.asm
...
```

```
index 0 = 3
index 1 = 4
index 2 = 4
index 3 = 2
index 4 = 1
```

Oops… it seems in this case we've got the value 4 twice, and no 0… Let's try with more numbers (0–255) and print the distribution and coverage.

```
.const NB_RAND=256
.var vals = List()
.for (var i = 0; i < NB_RAND; i++) {
    .eval vals.add(floor(random() * NB_RAND))   // Compute once, store in a list
}

random_lut:
    .fill NB_RAND, vals.get(i)                   // Fill from the list

.print ""
.print "Values:"
.for (var i = 0; i < NB_RAND; i++) {
    .print "index " + i + " = " + vals.get(i)   // Print the values
}

.print ""
.print "Distribution:"
.var counts = List()
.for (var v = 0; v < NB_RAND; v++) {
    .eval counts.add(0)                          // One counter per possible value
}
.for (var i = 0; i < vals.size(); i++) {
    .var v = vals.get(i)
    .eval counts.set(v, counts.get(v) + 1)
}

// print a histogram row per value
.for (var v = 0; v < NB_RAND; v++) {
    .var n = counts.get(v)
    .var bar = ""
    .for (var b = 0; b < n; b++) {
        .eval bar = bar + "#"
    }
    .print "value " + v + " | " + bar + " (" + n + ")"
}

.print ""
.print "Coverage:"
.var seen = Hashtable()
.for (var i = 0; i < NB_RAND; i++) {
    .eval seen.put(vals.get(i), true)
}
.print "Covered " + seen.keys().size() + " of "+NB_RAND+" cells"
```

Listing 1: 01_precomputed-random.asm

Here is the output:

```
Values:
index 0 = 71
index 1 = 153
index 2 = 54
index 3 = 48
...
index 252 = 250
index 253 = 166
index 254 = 190
index 255 = 15

Distribution:
value 0 |  (0)
value 1 | ## (2)
value 2 |  (0)
value 3 | ## (2)
value 4 |  (0)
value 5 | #### (4)
value 6 | # (1)
...
value 254 |  (0)
value 255 | ## (2)

Coverage:
Covered 163 of 256 cells
```

That's not going to cut it... let's check the coverage for what we actually need to fill the screen: 1000 cells.

```
.const NB_RAND=1000

... same code ...
```

```
Coverage:
  Covered 633 of 1000 cells
```

That means only 633 of the 1000 cells we need will actually be filled. We don't need an array of 1000 random numbers... we need a permutation of 1000 distinct numbers.

// Precomputed LUT of permutations

We can start from here:

```
.var lutList = List()
.for (var i = 0; i < 1000; i++) {
    .eval lutList.add(i)      // Add numbers 0 to 999 to a list
}
.eval lutList.shuffle()       // And shuffle that list

.pc = * "1000 Shuffled Random Values LUT"
permutation_lut:
    .lohifill 1000, lutList.get(i)  // Fill memory from the shuffled list
```

Good, now we can display them on the screen. We have a list of 1000 entries where the individual values are between 0 and 999. That means we need a 16-bit pointer to iterate over the list, and the values in the list are also 16-bit…

Do you remember that

```
permutation_lut:
    .lohifill 1000, lutList.get(i)  // Fill memory from the shuffled list
```

This is the same as this. Let's keep that syntax for the explanation.

```
permutation_lut_lo:
    .fill 1000, <lutList.get(i) // Lo byte
permutation_lut_hi:
    .fill 1000, >lutList.get(i) // Hi byte
```

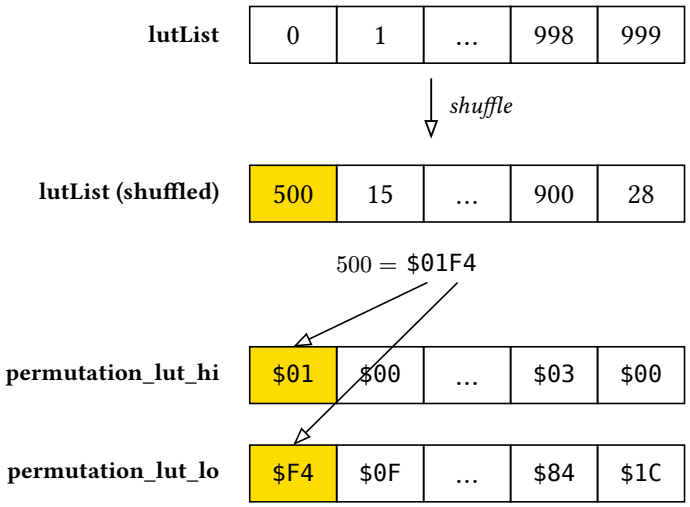


Figure 2: An ordered list `0…999` is shuffled into a random permutation, then each value is split into its high and low bytes across two parallel tables (`500 = $01F4` ⟶ hi `$01` into `permutation_lut_hi`, lo `$F4` into `permutation_lut_lo`).

> **Side note:** As you can see, we are going to use 16-bit extensively. If you are not yet familiar with it. you can have a look at the "16-Bit Pointer Arithmetic Cheat Sheet" at the end of this part.

That means our program should write its first filled char at $0400 (for the screen base address) + $01F4 = $05F4. Here is a program that does exactly this: it reads a single position from a LUT and places a filled char at that spot on the screen.

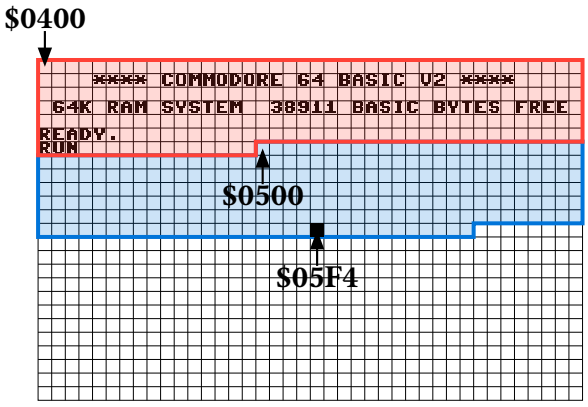


Figure 3: Drawing a filled char at position $05F4

```
.pc = $02 virtual
target_ptr: .word 0
lut_read_lo_ptr: .word 0
lut_read_hi_ptr: .word 0

BasicUpstart2(main)

main:
    // Initialize read pointers
    // for the lut_lo table
    // to start of tables
    lda #<lut_lo
    sta lut_read_lo_ptr
    lda #>lut_lo
    sta lut_read_lo_ptr + 1

    // same for lut_hi
    lda #<lut_hi
    sta lut_read_hi_ptr
    lda #>lut_hi
    sta lut_read_hi_ptr + 1

    // Fetch target screen address
    // from lut_lo/lut_hi
    // using our read pointers
    ldy #0
    lda (lut_read_lo_ptr), y
    sta target_ptr
    lda (lut_read_hi_ptr), y
    sta target_ptr + 1
    // Add base screen address ($0400)
    lda target_ptr+1

    // adding HIGH byte ($04) only
    // is ok as LOW byte is $00
    clc
    adc #>$0400
    sta target_ptr+1

    lda #160   // filled char
    sta (target_ptr), y

forever:
    jmp forever

// LUT to hold our 1000 values
// using a single hard-coded value
// for $01F4
lut_lo:
    .byte $F4
    .fill 999, 0 // not used for now
lut_hi:
    .byte $01
    .fill 999, 0 // not used for now
```

Listing 2: 02_single_char.asm

The 16-bit pointers are the hardest part. If you are not used to them yet, you should stop here for a while, until you really get what's going on.

> **Side note:** "I'm lost... why not just do this and be done with it?" you may think.
>
> ```
> lda #160
> sta $0400 + $01F4
> ```
>
> Yes, that does the same thing... but how do you move on from there? Using self-modifying code? That won't be any simpler. The plumbing we saw in the previous listing is all about preparing for the next step, which is to fill the whole screen with random positions.

Here is the whole source code for the full implementation, which fills the entire screen (03_precomputed-permutations.asm).

```
.pc = $02 virtual
// target_ptr for both
// the screen at $0400
// and the coloram at $d800
target_ptr: .word 0
```

```
permutations_lo_ptr: .word 0
permutations_hi_ptr: .word 0
current_index: .word 0
save_target_ptr_hi: .byte 0

BasicUpstart2(main)

//  Compile-Time Shuffled
// Table Generation (0 to 999)
.eval var lutList = List()
.for (var i = 0; i < 1000; i++) {
    .eval lutList.add(i)
}
.eval lutList.shuffle()

main:
    lda #<permutations_lut.lo
    sta permutations_lo_ptr
    lda #>permutations_lut.lo
    sta permutations_lo_ptr + 1

    lda #<permutations_lut.hi
    sta permutations_hi_ptr
    lda #>permutations_lut.hi
    sta permutations_hi_ptr + 1

    lda #0
    sta current_index
    sta current_index + 1

main_loop:
    // number of chars to display at once
    // during a single frame
    ldx #8 // 8 chars per frame.

single_frame_loop:
    // Process ONE random position
    jsr fill_next_screen_pos

    // Check if counter == 1000
    lda current_index + 1
    cmp #>1000
    bne continue
    lda current_index
    // Stop when all 1000 positions
    // are plotted
    cmp #<1000
    beq done
continue:
    dex
    bne single_frame_loop

    jsr delay
    jmp main_loop
```

```
done:
    // flash border color when done
    !:
    inc $d020
    jmp !-

fill_next_screen_pos:
    ldy #0

    // Fetch target screen address from LUT pointers
    lda (permutations_lo_ptr), y
    sta target_ptr
    lda (permutations_hi_ptr), y
    sta target_ptr + 1

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

    // Plot character 160 to target screen RAM
    lda #160
    sta (target_ptr), y

    lda save_target_ptr_hi
    clc
    adc #>$d800
    sta target_ptr+1
    lda #BLACK
    sta (target_ptr), y


    // Increment LOW table read pointer
    inc permutations_lo_ptr
    bne !+
    inc permutations_lo_ptr + 1
!:
    // Increment HIGH table read pointer
    inc permutations_hi_ptr
    bne !+
    inc permutations_hi_ptr + 1
!:

    // Increment 16-bit item counter
    inc current_index
    bne !+
    inc current_index + 1
!:
    rts

delay:
   // slow down filling the screen by waiting
   // for the raster line to be at the top
```

```
    // of the screen
    bit $d011
    bpl *-3
    bit $d011
    bmi *-3
    rts

.pc = * "Permutation LUT, 1000 Shuffled Random Values"
permutations_lut:
    .lohifill 1000, lutList.get(i)
```

This is our longest listing so far.

Is it the way to go? Probably not.

This code works fine, but **we are using 2000 bytes on the permutation LUT**. That's a lot.

> **Side note:** 2000 bytes, Is that really a lot?
>
> It always depends, but on a Commodore 64, burning 2 KB just to shuffle screen positions is a heavy tax.
>
> LUTs are great when you need raw speed, but in our case, it makes for a fun exercise to see if we can fill the screen without sacrificing all that RAM.

// Next challenge

We will continue on this, and try to avoid using 2000 bytes. Could you write an implementation that does mostly the same, but without using precomputed values?

// Recap of the mnemonics we used so far

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 ($M = M + 1$).
`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.
`tya`	Transfer Y to Accumulator — Copies the value from register Y into register A ($A = Y$). Sets the Zero flag ($Z = 1$) if the value is 0 and the Negative flag ($N = 1$) if bit 7 is set.
`inx`	Increment X Register — Increases register X by 1 ($X = X + 1$).
`bit`	Bit Test. Performs a bitwise AND between the memory contents and the Accumulator without altering either register. Sets the Zero flag (Z) based on the AND result, and copies memory bit 7 into the Negative flag (N) and memory bit 6 into the Overflow flag (V).

New Mnemonics introduced in Round 4

Mnemonic	Description
`sec`	Set Carry Flag — Sets the Carry flag ($C = 1$), required before 8-bit or 16-bit subtraction (`sbc`).
`sbc`	Subtract with Carry — Subtracts a value and the borrow from register A ($A = A - M - (1 - C)$). Set the Carry with `sec` first so the initial borrow is 0.

// 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

- Discuss a solution to our challenge that doesn't waste 2000 bytes.

// Kick Assembler: 16-Bit Pointer Arithmetic Cheat Sheet

ZP Pointer Setup & Initialization

Define the pointer in virtual ZP memory and initialize it with an address.

```
.pc = $02 "ZeroPage" virtual
zp_pointer: .word $0000

// Initialize with $0400
lda #<$0400
sta zp_pointer
lda #>$0400
sta zp_pointer+1
```

Indirect Indexed Addressing `(ZP),Y`

Read or write data at `[zp_pointer] + Y`.

```
ldy #0
lda (zp_pointer),y   // Read byte at pointer
sta (zp_pointer),y   // Write byte to pointer

// NOTE: Only 'y' works for (zp),y
// Writing (zp_pointer),x in Kick Assembler
// silently compiles as 'sta zp_pointer,x'
// and corrupts your ZP pointer!
```

Increment Pointer (+1)

Inc low byte; if it rolls over to `$00`, inc high byte using anonymous forward label `!+`.

```
inc zp_pointer       // Inc low byte
bne !+               // Skip if no overflow
inc zp_pointer+1     // Inc high byte
!:                   // Anonymous label target
```

Decrement Pointer (-1)

Check low byte before decrementing to borrow from high byte.

```
lda zp_pointer       // Check low byte
bne !+               // Skip borrow if low != 0
dec zp_pointer+1     // Borrow 1 from high byte
!:
dec zp_pointer       // Dec low byte
```

Add 8-bit Offset to Pointer

Add an 8-bit offset (e.g., +40 for a C64 row) to `zp_pointer`.

```
clc
lda zp_pointer
adc #40              // Add 8-bit offset
sta zp_pointer
bcc !+               // Skip high byte if no carry
inc zp_pointer+1
!:
```

Add 16-bit Value to Pointer

Add a 16-bit constant (e.g., +1000 = `$03e8`) using '<' and '>'.

```
clc
lda zp_pointer
adc #<$03e8          // Add low byte
sta zp_pointer
lda zp_pointer+1
adc #>$03e8          // Add high byte with carry
sta zp_pointer+1
```

Subtract 8-bit Offset from Pointer

Subtract an 8-bit offset (e.g., −40) from `zp_pointer`.

```
sec                  // Set carry = no borrow
lda zp_pointer
sbc #40              // Sub 8-bit offset
sta zp_pointer
bcs !+               // Skip high dec if no borrow
dec zp_pointer+1
!:
```

Subtract 16-bit Value from Pointer

Subtract a 16-bit constant (e.g., −1000 = `$03e8`) from `zp_pointer`.

```
sec                  // Set carry = no borrow
lda zp_pointer
sbc #<$03e8          // Sub low byte
sta zp_pointer
lda zp_pointer+1
sbc #>$03e8          // Sub high byte with borrow
sta zp_pointer+1
```

Kick Assembler Anonymous Labels

- `!:` Reusable anonymous label target.
- `!+` Branch **forward** to the next `!:`.
- `!-` Branch **backward** to the previous `!:`.

Key Rules

- Addition: **`clc`** before `adc` (Carry = 1 indicates overflow).
- Subtraction: **`sec`** before `sbc` (Carry = 0 indicates borrow).
- < extracts low byte, > extracts high byte.