16.

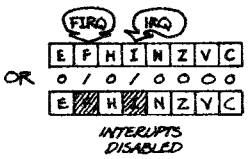
Indirect addressing is coming along. There's still color graphics mode details to deal with, and some ideas in quickly moving data from place to place. Clock cycles will come into play in the evaluation of speed -- since we all want graphics and speed to go hand in hand. There are also reviews of former information to cover, so that this programming doesn't move along too fast.

Although we're still pretty far from its actual use in this program, I want to remind you that the current topic is indexed indirect addressing. Indexed indirect is the mode where the operand is the address of a memory location, and the contents of the pair of sequential 8-bit memory locations make up an address which is the eventual location of the data. For example, say register X points to memory location **$3000**. Say that memory location **$3000** contains byte **$AB** and memory location **$3001** contains byte **$99**. Now say finally that memory location **$AB99** contains byte **$FF**. Load A accumulator zero-offset indexed indirect to X would result in A containing **$FF**. I've got some illustrations of that concept in the book, and the program we're writing should help understand the usefulness of the technique.

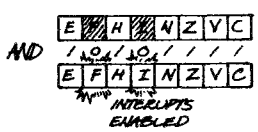
By this session you should have prepared the setup information and the scratchpad memory clearing. Your setup should look something like mine. To disable interrupts, you would . . .

```
ORCC   #$50
```

. . . which sets bits 4 and 6 of the condition code register.



To choose color set #1, you need to set bit 3 of port address **$FF22**. Furthermore, bits 4, 5 and 6 are the graphics mode selection bits, and bit 7 is the alphanumeric/graphic selector. If you hadn't taken a look at all these control bits, then now's the time. Turn to page 15 of your MC6883 data booklet (page 15 of the SAM data booklet).



There are 16 different display modes presented on this page, all but one available in two color sets. That gives you 31 choices. This wide selection is only available in computers where the 6847 video display generator and the 6883 SAM are used together; both are smart circuits, and so they interact in complex and versatile ways. The mode I've selected for the Game of Life is full color graphics 3. If you follow down on the chart, you'll see full graphics 3C, and the required bit conditions. The detailed memory map shows these bits; I'll remind you that the MC6847 modes

```
* What addressing mode is LDX
#1234?

Extended addressing.

* What addressing mode is LDX
(#1234)?

Indirect addressing.

* What specific addressing mode
is LDX #1234?

Extended addressing.

* What specific addressing mode
is LDX (#1234)?

Extended indirect addressing.
```

* What specific addressing mode is LDA ,Y?

Zero-offset indexed.

* What specific addressing mode is LDA (,Y)?

Zero-offset indexed indirect addressing.

* What is necessary to choose VDG color set #1 on the Color Computer?

Setting bit 3 of port address $FF22 chooses color set #1.

* Which bits of port $FF22 select the graphics modes?

Bits 4, 3 and 6 select the graphics modes.

* Which bit of port $FF22 selects between alphanumerics and graphics?

Bit 7 of port $FF22 selects alphanumerics or graphics.

* Name my three cats.

Aida, Mehitabel and Beulah.

* There are basic rules to the Game of Life. What is the general term for a set of rules?

An algorithm.

* There are four parts to the Game of Life algorithm, each involving a cell position on a grid. How many potential neighbors are there in a regular, two-dimensional square grid?

Eight neighbors.

* What do three neighbors produce in a dormant cell?

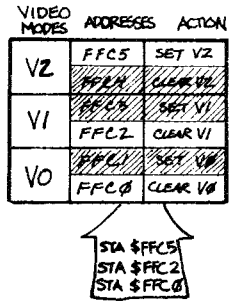
A cell "birth".

(the first five columns) are, respectively, bits 7, 6, 5, 4 and 3.

So to achieve mode G3C, bit 7 must be high, bit 6 is high, bits 5 and 4 are low, and bit 3 is up to you. Bit 3 is high for color set #1. So the left five bits of the binary number created for port **$FF22** is **11001**. The rightmost three digits are powered up to 111 on a 16K machine. So the complete binary number to select full color graphics mode 3, color set 1, is **11001111**, or hex **$CF**. The instructions are simplicity itself . . .

```
LDA     #$CF
STA     $FF22
```

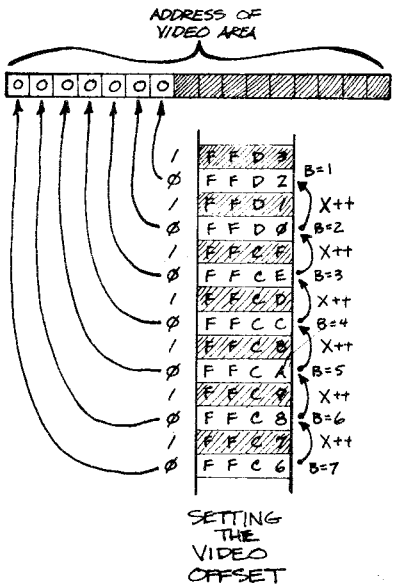
. . . easily selecting the proper modes.



But in the process, don't forget the SAM. It has to be properly programmed as well. According to the chart you've been looking at, mode G3C requires that SAM bits V2, V1 and V0 be programmed binary 100. Turn the page in the SAM data booklet, and look at the map on page 17. Addresses **$FFC0** through **$FFC5** control the SAM modes. To set mode G3C, then, set bit V2 and clear bits V1 and V0. That means, remembering the SAM's write-only register technique, write to addresses **$FFC5**, **$FFC2**, and **$FFC0**. So you store any value to **$FFC5**, **$FFC2** and **$FFC0** . . .

```
STA $FFC5
STA $FFC2
STA $FFC0
```

The final setup information is to choose the display memory starting address, which I've selected to be found at **$0000**. The display offset information is provided by the SAM, so that setup information must be written to the SAM. That is done — again recalling one of early lessons — by writing to addresses **$FFC6** through **$FFD2**. To establish starting address **$0000** means that binary values **0000 000** must be put into the seven SAM address offset positions. To place a zero in the SAM — that is, to clear a bit — you write to an even-numbered address. To place a one in the SAM — to set a bit — you write to an odd-numbered address. As you've seen, the display memory calls for binary **0000 000**, so that calls for writing to addresses **$FFC6**, **$FFC8**, **$FFCA**, **$FFCC**, **$FFCE**, **$FFD0** and **$FFD2**.

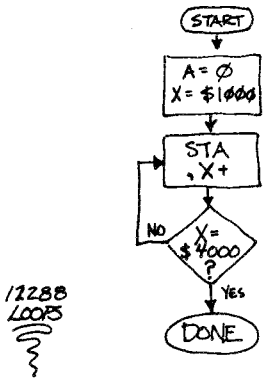


The most straighforward way of doing this might be to store an accumulator at each location . . . **STA $FFC6**, **STA $FFC8**, **STA $FFCA**, etc. Since all even addresses are being set, I've chosen this case-specific solution. . .

```
        LDB  #$07
        LDX  #$FFC6
VIDEO   STA  ,X++
        DECB
        BNE  VIDEO
```

... which writes to every other even-numbered address for a total of seven. The B register does the counting, and the X register points to the first SAM video address. For contrast, have a look at the general-purpose example shown at the bottom of page 16 in the SAM data booklet.

This completes the pre-program setup; take a break now and review this process, especially if your solutions are substantially different from mine.

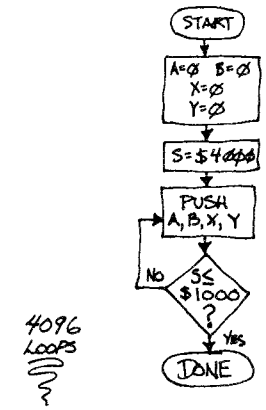


12288 LOOPS

Last time I suggested you have a look at stack instructions and see how they might be used to fill memory. First, here's a standard method of doing a memory fill from **$1000** to **$3FFF** (you can follow along in the book):

```
        CLRA            * Set A to Ø
        LDX   #$1ØØØ    * Point X to ;b;$1ØØØ
LOOP    STA   ,X+       * Store zero
        CMPX  #$4ØØØ    * See if finished
        BNE   LOOP      * Go back til done
```

The main time-consuming part of this routine consists of the last three instructions, requiring 6, 4 and 3 cycles respectively. The total of 13 cycles is repeated 12,288 times, for a total of 159,744 cycles. At 894,886 clock cycles per second, this operation takes a considerable 178.5 milliseconds... nearly one-fifth of a second. For fast action games, that isn't.



4096 LOOPS

Consider this solution instead (follow me in the book):

```
        CLRA            * Set A to Ø
        CLRB            * Set B to Ø
        TFR   D,X       * Set X to Ø
        TFR   D,Y       * Set Y to Ø
        LDS   #$4ØØØ    * Point S to top
LOOP    PSHS  A,B,X,Y   * Push 6 bytes
        CMPS  #$1ØØØ    * See if bottom
        BPL   LOOP      * Back until lower
```

After clearing A, B, X and Y to zero, the S stack is pointed to the top of the memory area to be cleared. Remember that the stack pushes down from the top. A, B, X and Y are then pushed on the stack using one instruction. The stack is compared immediate with **$1000**, the bottom of scratchpad memory, and if the result is plus (if S is greater than or equal to **$1000**), the routine is repeated.

The number of clock cycles required for the PSHS instruction is 5 plus 1 additional for each byte pushed. Six bytes are pushed in total, meaning that PSHS A,B,X,Y takes 11 cycles to complete. So the heart of this memory fill routine requires 11, 5 and 3 cycles... a total of 19. 19 cycles is longer than the 13 needed for the previous example. But remember that in this case, *six* bytes are pushed at once.

* What do two or three neighbors produce in a live cell?

No change.

* What do less than two neighbors produce?

A cell "death".

* What do more than three neighbors produce?

A cell "death".

* What addresses control the SAM video modes?

Addresses $FFC0 through $FFC5.

* What addresses control the SAM video display offset address?

Addresses $FFC6 through $FFD3.

* What is the video display offset address the address of?

The upper-left-most picture element shown on the video display screen.

* What is the term for "picture element"

Pixel.

* In the most detailed mode, what is the pixel size of the video screen (pixels wide by pixels high)?

256 pixels wide by 192 pixels high.

* What is the pixel size in mode CG3 (color graphics 3)?

2 pixels by 2 pixels.

* What is the size of the screen in mode CG3 (width by height)?

128 wide by 96 high.

Joe's neighborhood

* How many different points are displayed on the screen in mode CG3?

12,288 points.

* How many bytes are required for the screen in mode CG3?

3,072 bytes.

* How many colors are available in mode CG3?

Four colors.

* What are the four colors of VDG color set #0?

Green, yellow, blue and red.

* What are the four colors of VDG color set #1?

Buff, cyan, magenta and orange.

* The push and pull S stack commands require an operand, plus what additional information?

A postbyte.

* What information is contained in the postbyte?

Bits indicating which registers are to be pushed or pulled.

* How many registers can be pushed or pulled?

Eight registers.

* What are the eight registers which can be pushed or pulled?

PC, U, Y, X, DP, B, A and CC

* How many bytes are involved in pushing all eight registers?

12 bytes (2 each for PC, U, Y and X, 1 each for DP, B, A and CC).

12,288 divided by 6 is 2,048 . . . there are only 2,048 repetitions of this routine. 2,048 times 19 is 38,912 clock cycles; again, at 894,886 clock cycles per second, this instruction completes in only 43.5 milliseconds. That's slightly less than one-quarter the time of the previous method, just one-twentieth of a second.

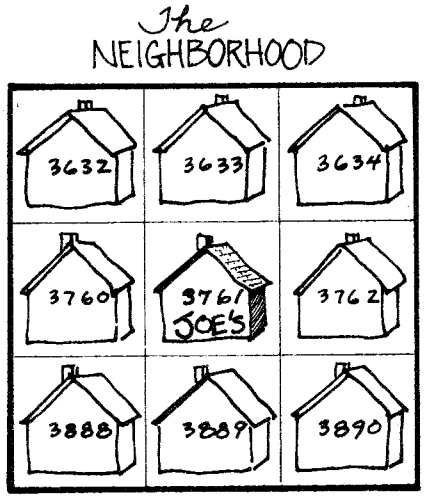
So where you need to fill blocks of memory very quickly, the push-stack method is ideal. Don't forget to save the stack pointer if you need to, and also to replace the stack pointer when you're done with it. In this program, I put the stack pointer at **$0DBF** when I'm finished with it:

```
LDS  #$0DBF
```

If you would like to look through these examples, this is a good time to stop and do that.

At this point, the Garden of Eden is populated, video display setup is complete, interrupts are disabled, scratchpad memory is cleared, and the stack is in position. It's time to evaluate the Garden of Eden for the population of its neighborhoods. I'll summarize the Hooper technique.

If Joe lives in a house on this regular memory grid, then he's potentially got a neighbor to the northwest, north, and northeast; to the west and east; and to the southwest, south, and southeast. Eight neighbors in all. The screen grid in this graphics mode is 128 by 96, 128 houses across by 96 houses down. If Joe lives in house #3761, then he's got potential neighbors in houses 3761 minus 129 (that's northwest), 3761 minus 128 (that's north), 3761 minus 127 (that's northeast). There's a house to the west at 3761 minus 1, and a house to the east at 3761 plus 1. Finally, there are houses to the southwest at 3761 plus 127, to the south at 3761 plus 128, and to the southeast at 3761 plus 129.



I'll convert those to hex. The screen is hex **$80** by **$60**, so Joe's got neighbors at **–$81**, **–$80**, **–$7F**, **–$01**, **+$01**, **+$7F**, **+$80**, and **+$81**. If the Y index register points to Joe, then the neighbor offsets would be:

-81	-80	-7F
-1	0	+1
+7F	+80	+81

```
        -$81,Y
        -$80,Y
        -$7F,Y
          -1,Y
           1,Y
         $7F,Y
         $80,Y
and      $81,Y
```

The process, then, is really a kind of inverse of this. If those eight are Joe's potential neighbors, then Joe is the neighbor of those eight. So instead of going to every cell and evaluating all eight neighbors, you can go to every cell and see if it is alive (that's the key). If it's alive, you increment the neighbor count; if not, you move along to the next.

So instead of making 12,288 checks of 8 neighbors, you make 12,288 checks for life. So only if a cell is live does the action become:

```
INC    -$81,Y
INC    -$80,Y
INC    -$7F,Y
INC      -1,Y
INC       1,Y
INC     $7F,Y
INC     $80,Y
INC     $81,Y
```

This is the heart of the neighborhood scratchpad routine. But think back to the actual display screen. Only 3,072 bytes are used to display the 12,288 cells. Somehow you've got to break these into quarter bytes very quickly, and evaluate them. You might choose a mask-and-shift strategy, where each pair of bits is shifted left, then masked and evaluated. You might choose a mask-and-compare strategy, where four separate routines are used to evalute the four separate quarter-bytes. Both methods would work, but in addition to masking or shifting, each technique would require saving and restoring the original value, testing, and branching.

0 0	DORMANT
0 1	NEWBORN
1 0	"DEVIANT"
1 1	MATURE

The method I've selected takes advantage of the rotate instruction, which rotates the bits of a byte around in a circle — but through the carry flag. You can take a look at the MC6809E data booklet to see exactly how the ROL and ROR instructions work. The advantage here is that, by carefully selecting how I represent live and dormant cells, I can rotate bits of the display byte through the carry flag and use the carry to branch to the proper routine. Rather than spend time explaining the concept, I'll take you right to the routine itself. Look in your book. As you examine the program excerpt, keep in mind that I've defined **00** as a dormant cell, **01** as a newborn cell, and **11** as a mature cell

* How many clock cycles does a push or pull operand use?

5 cycles.

* How many additional clock cycles are required for each register pushed or pulled?

1 cycle for each register pushed or pulled.

* How many cycles are required to execute the instruction PSHS PC,U,Y,X,DP,B,A,CC ?

17 cycles are required.

* How long is one Color Computer clock cycle?

1.11746 microseconds.

* How long does the instruction PSHS PC,U,Y,X,DP,B,A,CC take on the Color Computer?

About 19 microseconds (18.99 microseconds).

* How long would it take to fill 6,144 bytes of memory using PSHS PC,U,Y,X,DP,B,A,CC at 18.99 microseconds per instruction?

9723 microseconds (.009723 seconds), or about 1/100 of a second.

* How many clock cycles is PSHS A,B,X,Y?

5 plus 6, or 11.

* How long is PSHS A,B,X,Y?

11 times 1.11746, or 12.3 microseconds.

* How long would filling 6,144 bytes of memory take using PSHS A,B,X,Y?

19446 microseconds (.019446 seconds), or about 1/50 of a second).

Rotate to test

* It is theoretically possible to fill memory merely by executing a long series of PSHS A,B,X,Y. However, a comparison and branch would be required, resulting in a sequence like this:

```
LOOP   PSHS A,B,X,Y
       CMPS #$2000
       BPL  LOOP
```

How long would one iteration of this sequence take?

11 cycles plus 5 cycles plus 3 cycles, or 19 cycles total; 19 cycles times 1.11746 microseconds is 21.23 microseconds.

* How long would it take the above sequence to fill 6,144 bytes?

(21.23 microseconds per loop) times (6,144 bytes divided by 6 bytes per loop) is about 21,739 microseconds, about .02 seconds.

* The Game of Life uses 12,288 bytes. How long would it take to fill 12,288 bytes using this kind of sequence?

About .043 seconds.

* If the S stack pointer is set to $1000 and the instruction PSHS A,B,X,Y is executed, where will the S stack pointer be at the conclusion of the instruction?

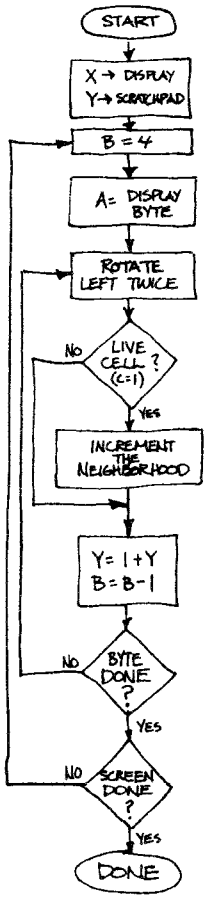
At $1000 minus 6, or $0FFA.

* Why $0FFA instead of $1006?

Because the stacks move downward in memory; they are push-down stacks.

(that is, past the first generation).

```
        LDX   #$0000   * Point X to display
        LDY   #$1000   * Point Y to scratchpad
NXTCEL  LDB   #4       * Count four quarter bytes
        LDA   ,X+      * Get video display byte
        PSHS  CC       * Save carry flag info
QUARTR  PULS  CC       * Restore carry flag info
        ROLA           * Rotate A through carry
        ROLA           * And rotate A again
        PSHS  CC       * Save carry (part of byte)
        BCC   NEXTQ    * If C=0, then cell not live
        INC   -$81,Y   * Northwest neighbor
        INC   -$80,Y   * Northern neighbor
        INC   -$7F,Y   * Northeast neighbor
        INC   -$01,Y   * Western neighbor
        INC   +$01,Y   * Eastern neighbor
        INC   +$7F,Y   * Southwest neighbor
        INC   +$80,Y   * Southern neighbor
        INC   +$81,Y   * Southeast neighbor
NEXTQ   LEAY  1,Y      * Get next scratchpad position
        DECB           * Count down quarter bytes
        BNE   QUARTR   * Get next quarter byte
        PULS  CC       * Else restore stack info
        CMPX  #$0C00   * See if at end of display
        BNE   NXTCEL   * Get next value
```



X and Y are pointed to display and scratchpad, respectively. The B accumulator serves as a quarter-byte counter. The A accumulator holds the byte from display to be evaluated.

Now here's the trick. The value in A is rotated left twice, through the carry flag. That leaves the rightmost bit of the display pair sitting in the carry flag, and the leftmost bit of the pair sitting in the bit **0** position of the accumulator. If the carry is clear, the cell is either dormant (**00**) or defined as illegal (**10**); in either case, it is not a neighbor, so the routine moves down to the label NEXTQ. If the carry is set, then the cell is either a newborn (**01**) or a mature cell (**11**), and the eight neighborhood incrementing instructions are completed. The label NEXTQ follows. More about the instruction "Load effective address" later; what you see effectively increments Y by one. The quarter-byte counter is decremented, and the rotate-and-branch routine is repeated until four quarter bytes have been done. The **CMPX #$0C00** tests for the end of the 3,072 byte display area. That's it. At the end of 3,072 groups of four quarter-byte tests, 12,288 bytes of scratchpad memory will be filled with neighborhood information.

To understand this process more intimately, take some time to draw a small grid of display points excerpted from the screen (say 16 by 16), a corresponding page of memory bytes (it would be 4 by 16), and a chart of scratchpad memory. Put some random cells in place on the display grid, then determine the display bytes. Finally, evaluate the results in scratchpad memory. In the next lesson, you'll do the actual neighborhood checking and updating.

17.

Hello again. I hope you aren't impatient with this step-by-step approach. In this lesson, you'll finally be getting to the application of indexed indirect addressing, and be completing the Game of Life. The result will be surprisingly short — under 240 bytes — and quite fast.

We left off having performed all the setups: disabling interrupts, selecting color graphics mode 3 with color set 1 (12,288 points in buff, cyan, magenta, orange), video display address **$0000**. 12,288 bytes of scratchpad memory has been cleared and filled with neighborhood information, that is, values 0 to 8.

Once the neighborhood values have been determined, that information is used to give birth to a cell, to allow a cell to become dormant, or to leave the cell unchanged. As with all programming, there are many ways to make this happen. And, as always, the most obvious solution isn't necessarily the fastest or the most efficient. The obvious solution is something like this . . .

```
LDA    ,Y
BEQ    DEATH
DECA
BEQ    DEATH
DECA
BEQ    NO CHANGE
DECA
BEQ    BIRTH
DECA
```

. . . and so on. Another technique — and a fast one — would have started by filling the scratchpad memory with **$FE** instead of **$00**. In this circumstance, zero or one neighbors would result in the scratchpad value being left with **$FE** or **$FF**. Two neighbors would produce **$00** in the scratchpad, three neighbors would yield **$01**, and more than three neighbors would produce **$02**. A much quicker method, the final routine would look like this . . .

Creating longer programs like this one can be time-consuming "up front", but care taken at this stage will assure a partly functioning -- if not perfect -- result when you type EXEC. No, this Game of Life didn't work for me the first time. But the screen showed proper modes and colors, and there was generation-to-generation motion. It wasn't right, but there was enough to begin serious debugging. If you spend your first few hours creating structure, then outlining modules, and finally stringing the pieces together, chances are your program will begin to show evidence of life from your first EXEC.

* What instructions are used to turn interrupts on and off?

ORCC and ANDCC.

* Why does ORCC #$50 turn off interrupts?

Because setting bits 4 and 6 of the condition code register turns off interrupts; #$50 is 01010000, so ORCC #$50 sets bits 4 and 6 without altering the other six bits.

Indexed indirection

* What addressing mode is JMP $3456?

Extended addressing.

* What addressing mode is JMP A,X?

Indexed addressing.

* What addressing mode is JMP (A,X)?

Indexed indirect addressing.

* X points to $1234. A is set to 4. The memory locations $1230 through $123F contain $01 02 45 47 93 96 A2 01 11 F5 36 92 19 5E 22 00. Where is the program counter after JMP $1234?

PC is at $1234.

* X points to $1234. A is set to 4. The memory locations $1230 through $123F contain $01 02 45 47 93 96 A2 01 11 F5 36 92 19 5E 22 00. Where is the program counter after JMP ($1234)?

PC is at $9396.

* X points to $1234. A is set to 4. The memory locations $1230 through $123F contain $01 02 45 47 93 96 A2 01 11 F5 36 92 19 5E 22 00. Where is the program counter after JMP A,X?

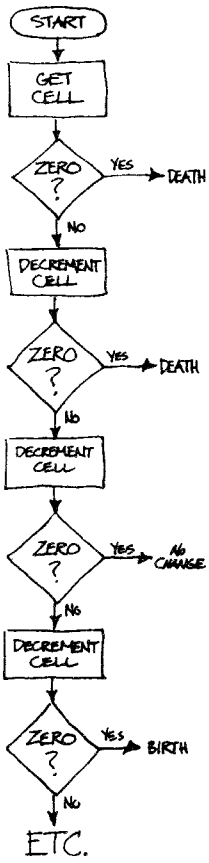
PC is at $1238.

* X points to $1234. A is set to 4. The memory locations $1230 through $123F contain $01 02 45 47 93 96 A2 01 11 F5 36 92 19 5E 22 00. Where is the program counter after JMP (A,X)?

PC is at $11F5.

```
        LDA     ,Y        * Get value from scratchpad
        BRN     DEATH     * If negative ($FE or ;b;$FF), then death
        BEQ     NO CHANGE* If zero ($00), then no change
        DECA              * Decrement A to set flags
        BEQ     BIRTH     * If zero ($01 minus 1), then birth
        BRA     DEATH     * Otherwise is ;b;$02 or greater
```

There's a lot in that short routine, and it's very fast. In fact, in this situation, it's a tossup in speed to the one I've chosen. Depending on the value in the scratchpad, it can take three, six, 11 or 14 clock cycles to complete; my sample method always takes 12 cycles. My guess is that in a "mature" civilization, the former method would be faster. But since this is a lesson on indirect indexed addressing, then indirect indexed addressing it is.



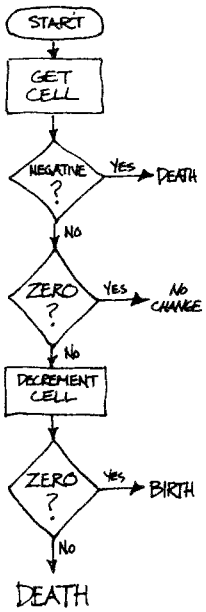
Indexed indirect looks like this . . .

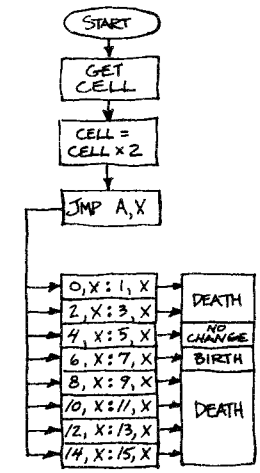
```
            ASLA              * Double A (two-byte offset)
            LDX     #TABLE    * Point to table of addresses
            JMP     [A,X]
        * Go to routine at X+A
TABLE       FDB     DEATH
            FDB     DEATH
            FDB     NO CHANGE
            FDB     BIRTH
            FDB     DEATH
```

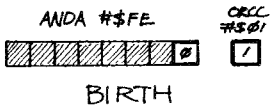
. . . etc. The A register is shifted left to double its value; this is true because an address is two bytes long and therefore requires a two-byte offset. The X register is pointed to the zeroeth entry in the table of jump addresses. The command JMP [A,X] causes the sum of A+X to be calculated, the data at addresses A+X and A+X+1 to be retrieved and concatenated, and the result to be given to the Program Counter. This routine is simple — more transparent than the earlier one — and demonstrates regularity and consistency. Take some time to review this routine and contrast it with the previous one. You'll note that where things might get complicated (for example, if there were ten or twenty choices instead of merely eight), the former routine gets serpentine and sluggish, whereas the indirect indexed jump is a fast and streamlined 6-byte jewel.

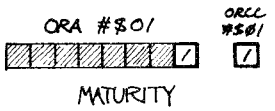
By the way. You see that I've used the notation "FDB" in the short program excerpt above. This is an assembler "pseudo-op", an instruction for the assembler to use the information you've provided and place the equivalent binary data in memory. The pseudo-op FCB places a single byte in the program; FDB places two bytes; and FCC places an ASCII string. Refer to the EDTASM+ manual, page 35, for details on how to use these.

I would like you to take a break here and examine the way this indexed indirect mode is used.









The scratchpad is being evaluated, so all that's left to write is the set of death, birth, and no-change routines. To force a cell into dormancy, both bits are set to zero; the resulting color is buff, the same color as the background. Recalling that the display byte has been rotated through the carry flag, the routine looks like this . . .

```
ANDA    #$FE    * Set to ØX
ANDCC   #$FE    * Set to ØØ
BRA     EXIT    * Go out
```

. . . and you can leave it to a general-purpose exit routine to complete the rotation and testing.

The no-change routine is slightly more complicated because it isn't really no change. As you recall, I wanted to add some visual variety by having newborn cells displayed in a different color from mature cells. Newborns are color **01** (cyan) and matures are 11 (orange), so "no change" for these means changing newborns to matures, and leaving matures as is. On the other hand, dormant cells are left dormant, and illegal cells present in the Garden of Eden are made dormant. Dormants are buff (**00**) and illegals are magenta (**10**), so "no change" for these means changing illegals to dormants, and leaving dormants as is. Here's how it looks:

```
        BCS     HIGH    * Go if C = 1
                        * C = Ø
        ANDA    #$FE    * Set to ØØ
        BRA     EXIT    * Go out
HIGH                    * C = 1
        ORA     #$Ø1    * Set to 11
        BRA     EXIT    * Go out
```

That leaves only the birth routine, which, if a cell is already alive, can be considered a "no change" routine. It is slightly more complex than the previous routines because dormant cells must be changed to newborns (**00** to **01**); illegal cells must be changed to newborns (**10** to **01**); newborns from the previous generation must be changed to oldsters (**01** to **11**); and oldsters are left unchanged (**11** to **11**). Putting it in chart form helps; look in the book:

Birth Routine

Present cell:	Changes to:
ØØ (buff) (dormant)	Ø1 (cyan) (newborn)
1Ø (magenta) (illegal)	Ø1 (cyan) (newborn)
Ø1 (cyan) (newborn)	11 (orange) (mature)
11 (orange) (mature)	11 (orange) (mature)

The carry flag is again the determining factor. If the carry flag is clear (zero), a newborn is created; if the carry flag is set, an oldster is created (or maintained). Here's how that looks . . .

```
* X points to $1234. A is set to 4. The memory locations $1230 through $123F contain $01 02 45 47 93 96 A2 01 11 F5 36 92 19 5E 22 00. Where is the program counter after JMP $09,X?

PC is at $123D.

* X points to $1234. A is set to 4. The memory locations $1230 through $123F contain $01 02 45 47 93 96 A2 01 11 F5 36 92 19 5E 22 00. Where is the program counter after JMP ($09,X)?

PC is at $5E22.

* X points to $1234. A is set to 4. The memory locations $1230 through $123F contain $01 02 45 47 93 96 A2 01 11 F5 36 92 19 5E 22 00. Where is the program counter after JMP -2,X?

PC is at $1232.

* X points to $1234. A is set to 4. The memory locations $1230 through $123F contain $01 02 45 47 93 96 A2 01 11 F5 36 92 19 5E 22 00. Where is the program counter after JMP (-2,X)?

PC is at $4547.

* X points to $1234. A is set to 4. The memory locations $1230 through $123F contain $01 02 45 47 93 96 A2 01 11 F5 36 92 19 5E 22 00. Where is the program counter after JMP ,X?

PC is at $1234.

* X points to $1234. A is set to 4. The memory locations $1230 through $123F contain $01 02 45 47 93 96 A2 01 11 F5 36 92 19 5E 22 00. Where is the program counter after JMP (,X)?

PC is at $9396.
```

Scratchpad

* What is the term for an instruction for the assembler to use the information you've provided to place binary data in memory?

A pseudo-op.

* What kind of information does the pseudo-op FCB place in memory?

One byte.

* What kind of information does the pseudo-op FDB place in memory?

Two bytes.

* What kind of information does the pseudo-op FCC place in memory?

An ASCII string of characters.

* Hand assemble the following:

```
LDX   $1234
JMP   $A0D7
FCB   $A6
FDB   $00F3
```

$BE 12 34 7E A0 D7 A6 00 F3

* What is another name for a work area of memory?

A scratchpad.

* What is the scratchpad used for in this Game of Life?

To store neighborhood information.

* In this Game of Life, what bit pair represents a dormant cell?

Bit pair 00.

* In this Game of Life, what bit pair represents a newborn cell?

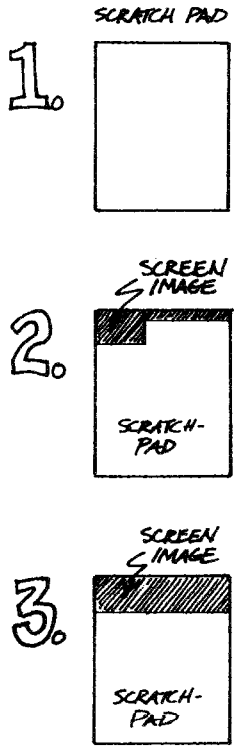
Bit pair 01.

* In this Game of Life, what bit pair represents a mature cell?

Bit pair 11.

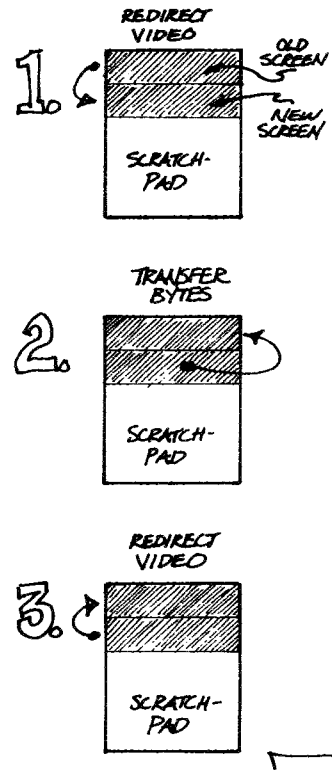
```
        BCC   LOW     * Pass if C = 0
                      * C = 1
        ORA   #$01    * Set to 11
        BRA   EXIT    * Go out
LOW                   * C = 0
        ANDA  #$FE    * Set to 00
        ORCC  #$01    * Set to 01
        BRA   EXIT    * Go out
```

So there you have the heart of it. There's some work to do right at the end. Consider this: if you store the display byte directly back on the screen, the new generation will swim down over the previous generation. Since one of the premises of the Game of Life is that all generational changes take place simultaneously, this swimming effect should be avoided. It can be avoided by filling a second area of memory and switching screens. But with 3,072 bytes required for display, 12,288 bytes required for the scratchpad, and about 230 bytes for program and stack, that leaves less than 700 bytes for a second screen. So what to do?



My solution lies in using that scratchpad for two purposes. Think of it this way. Each four-cell display byte is represented by four bytes of scratchpad memory. Once four scratchpad bytes have been used to determine the new display byte, they are no longer needed. After eight scratchpad bytes are evaluated, two display bytes have been produced. After all 12,288 scratchpad bytes have been used, 3,072 display bytes have been produced.

In that pattern lies the opportunity. The new display screen can be placed in scratchpad memory, because the using up of the scratchpad memory always outpaces by a ratio of four to one the production of display memory bytes. When the new screen has been produced, the video offset address in the SAM can be switched to that new screen in scratchpad memory.



Now since scratchpad memory has to be used again for the next generation, that screen has to be ushered out of that area of memory. Once the video has been redirected from **$0000** to **$1000**, the contents beginning at **$1000** can be transferred to memory beginning at **$0000**. Then, when the original display memory is filled with the new generation, the video offset address can be switched back. The evaluation of the generation, production of the new generation, screen switching, and display memory transfers are entirely invisible. Here's how the code looks..

```
        STA   $FFCD   * Switch to screen at $1000
        LDX   #$1000  * Point X to new screen
        LDY   #$0000  * Point Y to old screen
TRNSFR  LDA   ,X+     * Get value from new screen
        STA   ,Y+     * Store value to former screen
        CMPX  #$1C00  * See if screen is finished
        BNE   TRNSFR  * Go back to finish screen
        STA   $FFCC   * Redirect video to $0000
```

And there you have it. All that's really left to do is to put all the pieces together, keep track of where and how the stack is used, and organize the automatic repeating process to keep the generations going. The entire commented program listing follows on this tape, and is also printed in the book. You can load and examine this listing, and then run the object code which is also on the tape. After you're done reviewing the listing and trying the program, I'll summarize the programming concepts and ideas from these past three sessions.

Program #27A, an EDTASM+ program. Insert the EDTASM+ cartridge, and turn on the power to your computer. When the cursor appears, type L and press ENTER. The computer will search (s) and find (F). When the cursor reappears, display the program. Type P#:* and press ENTER. If the right-hand side of the program is not similar to the listing, or it an I/O error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

```
* In this Game of Life, what  is
bit pair 10?

An "illegal" or "deviant" cell.

* Why  is bit pair 10 an illegal
or deviant cell?

Because  it  is  present only in
the Garden of Eden, but  is  not
found in future generations.

* Why are bit pairs used?

Two     provide     a     four-color
display.

* What video mode is this?

Video mode CG3.
```

```
                        00100 ********************************************************
                        00110 *                                                      *
                        00120 *                  THE GAME OF LIFE                    *
                        00130 *                                                      *
                        00140 *    BASED ON THE PASTIME DEVELOPED BY JOHN CONWAY     *
                        00150 *                                                      *
                        00160 * COLOR COMPUTER VERSION 1.0 BY DENNIS BATHORY KITSZ *
                        00170 *                                                      *
                        00180 ********************************************************
                        00190 *
0C80                    00200          ORG      $0C80
                        00210 *
                        00220 * DISABLE INTERRUPTS BY MASKING I AND F BITS IN CCR  *
                        00230 *
0C80 1A   50            00240 START    ORCC     #$50     * DISABLE INTERRUPTS
                        00250 *
                        00260 * COLOR SET 0 CHOICES ARE GREEN, YELLOW, BLUE, RED
                        00270 * COLOR SET 1 CHOICES ARE BUFF, CYAN, MAGENTA, ORANGE
                        00280 * PORT ADDRESS $FF22 CONTROLS COLOR SETS, OTHER INFO
                        00290 *
0C82 86   CF            00300          LDA      #$CF     * VALUE FOR COLOR SET
0C84 B7   FF22          00310          STA      $FF22    * CHOOSE COLOR SET
                        00320 *
                        00330 * THE DISPLAY SCREEN MEMORY IS SELECTED TO RUN FROM
                        00340 * $0000 TO $0BF0 (3072 BYTES), USING COLOR GRAPHICS
                        00350 * MODE 3.  THE FOLLOWING ROUTINE RESETS EVEN ADDRESSES
                        00360 * $FFC6 THROUGH $FFD2 IN THE SAM, SELECTING THE VIDEO.
                        00370 *
0C87 C6   07            00380          LDB      #$07     * VIDEO DISPLAY ADDRESSES
0C89 8E   FFC6          00390          LDX      #$FFC6   * FIRST SAM VIDEO ADDRESS
0C8C A7   81            00400 VIDEO    STA      ,X++     * DO EVERY OTHER ADDRESS
0C8E 5A                 00410          DECB              * DONE WITH SETUP YET?
0C8F 26   FB            00420          BNE      VIDEO    * DO NEXT DISPLAY ADDRESS
                        00430 *
                        00440 * THERE ARE THREE GRAPHICS MODES TO BE SELECTED TO
                        00450 * ACHIEVE COLOR GRAPHICS MODE 3.  SAM ADDRESSES $FFC0
                        00460 * THROUGH $FFC5 SET UP COLOR GRAPHICS MODES.
                        00470 *
0C91 B7   FFC5          00480          STA      $FFC5    * SET GRAPHIC MODE 2
0C94 B7   FFC2          00490          STA      $FFC2    * RESET GRAPHIC MODE 1
0C97 B7   FFC0          00500          STA      $FFC0    * RESET GRAPHIC MODE 0
                        00510 *
                        00520 * THE FOLLOWING ROUTINE CLEARS A 12K AREA OF MEMORY
                        00530 * FOR USE AS A SCRATCHPAD WORK AREA WHEN EVALUATING
                        00540 * THE PRESENT GENERATION OF CELLS.  THE METHOD CHOSEN
                        00550 * HERE TO CLEAR MEMORY IS VERY FAST.  INSTEAD OF A
                        00560 * "STA ,X+" STYLE OF MEMORY FILLING, SIX BYTES (THE
                        00570 * TWO ACCUMULATORS PLUS THE X AND Y REGISTERS) ARE
                        00580 * CLEARED TO ZERO.  THE STACK IS POINTED TO THE TOP
                        00590 * OF THE MEMORY TO BE CLEARED, AND THE SIX BYTES ARE
                        00600 * PUSHED ON THE STACK UNTIL THE MEMORY AREA IS FULL.
                        00610 *
```

```
0C9A 4F             00620 AGAIN   CLRA              * SET ACCUMULATOR A = 0
0C9B 5F             00630         CLRB              * SET ACCUMULATOR B = 0
0C9C 1F   02        00640         TFR     D,Y       * SET REGISTER Y = 0
0C9E 1F   01        00650         TFR     D,X       * SET REGISTER X = 0
0CA0 10CE 4000      00660         LDS     #$4000    * STACK TO TOP OF SCRATCHPAD
0CA4 34   36        00670 NEXT1   PSHS    A,B,X,Y   * PUSH 6 BYTES ON THE STACK
0CA6 118C 1000      00680         CMPS    #$1000    * IS STACK UNDER #$1000 YET?
0CAA 2A   F8        00690         BPL     NEXT1     * KEEP GOING IF NOT THERE
                    00700 *
                    00710 * THE STACK IS THEN SET OUT OF THE WAY OF THE
                    00720 * "NEIGHBORHOOD" INCREMENTING ROUTINE WHICH FOLLOWS.
                    00730 *
0CAC 10CE 0DBF      00740         LDS     #$0DBF
                    00750 *
                    00760 * THE "NEIGHBORHOOD" INCREMENTING ROUTINE USES THE
                    00770 * HOOPER METHOD.  IN MOST CONCEPTUALIZATIONS OF THE
                    00780 * GAME OF LIFE, EACH CELL POSITION IS CHECKED FOR
                    00790 * THE NUMBER OF NEIGHBORS WHICH SURROUND IT.  IT TURNS
                    00800 * OUT THAT, AFTER THE FIRST GENERATION (THE GARDEN OF
                    00810 * EDEN GENERATION), THERE ARE ALWAYS LESS LIVE CELLS
                    00820 * THAN DORMANT ONES.  SO INSTEAD OF CHECKING FOR THE
                    00830 * NEIGHBORS OF EACH CELL, IT IS FASTER TO CHECK EACH
                    00840 * CELL TO DETERMINE WHOSE NEIGHBOR IT IS.  WHEN ALL
                    00850 * CELLS HAVE BEEN CHECKED, A COUNT OF NEIGHBORS HAS
                    00860 * BEEN CREATED.
                    00870 *
0CB0 8E   0000      00880         LDX     #$0000    * POINT X TO DISPLAY
0CB3 108E 1000      00890         LDY     #$1000    * POINT Y TO SCRATCHPAD
                    00900 *
0CB7 C6   04        00910 NXTCEL  LDB     #$04      * COUNT FOUR QUARTER BYTES
0CB9 A6   80        00920         LDA     ,X+       * GET VIDEO DISPLAY BYTE
0CBB 49             00930 QUARTR  ROLA              * ROTATE A THROUGH CARRY
0CBC 49             00940         ROLA              * ROTATE A THROUGH CARRY
0CBD 24   19        00950         BCC     NEXTQ     * IF C=0, THEN CELL NOT LIVE
                    00960 *
                    00970 * HERE IS THE NEIGHBORHOOD:
                    00980 *       -------------------
                    00990 *       / -81 / -80 / -7F /
                    01000 *       -------------------
                    01010 *       / -01 / JOE / +01 /
                    01020 *       -------------------
                    01030 *       / +7F / +80 / +81 /
                    01040 *       -------------------
                    01050 *
0CBF 6C   A9 FF7F   01060         INC     -$81,Y    * UPPER LEFT NEIGHBOR
0CC3 6C   A8 80     01070         INC     -$80,Y    * UPPER NEIGHBOR
0CC6 6C   A8 81     01080         INC     -$7F,Y    * UPPER RIGHT NEIGHBOR
0CC9 6C   3F        01090         INC     -$01,Y    * LEFT NEIGHBOR
0CCB 6C   21        01100         INC     $01,Y     * RIGHT NEIGHBOR
0CCD 6C   A8 7F     01110         INC     $7F,Y     * LOWER LEFT NEIGHBOR
0CD0 6C   A9 0080   01120         INC     $80,Y     * LOWER NEIGHBOR
0CD4 6C   A9 0081   01130         INC     $81,Y     * LOWER RIGHT NEIGHBOR
                    01140 *
0CD8 31   21        01150 NEXTQ   LEAY    1,Y       * GET NEXT SCRATCHPAD POSITION
0CDA 5A             01160         DECB              * COUNT DOWN BY QUARTER BYTES
0CDB 26   DE        01170         BNE     QUARTR    * GET NEXT QUARTER BYTE
0CDD 8C   0C00      01180         CMPX    #$0C00    * SEE IF END OF DISPLAY
0CE0 26   D5        01190         BNE     NXTCEL    * ELSE GET NEXT VALUE
                    01200 *
                    01210 * ONCE THE NEIGHBORHOODS HAVE BEEN DETERMINED, THE
                    01220 * INFORMATION IS USED TO GIVE BIRTH TO A CELL, ALLOW
                    01230 * A CELL TO DIE, OR LEAVE THE CELL UNCHANGED.  A TEST
                    01240 * FOR 0 OR 1; 2; 3; 4 OR MORE NEIGHBORS COULD BE DONE
                    01250 * BY USING THE SCRATCHPAD INFORMATION.  IN THIS CASE,
                    01260 * THE INFORMATION (0 THROUGH 8 NEIGHBORS) IN THE
                    01270 * SCRATCHPAD IS USED AS AN OFFSET TO A TABLE OF
                    01280 * ADDRESSES.  THE X REGISTER POINTS TO THE ZEROETH
                    01290 * ENTRY IN THE TABLE, AND THE A REGISTER PROVIDES THE
                    01300 * OFFSET.  X+A IS THE ADDRESS OF THE DEATH, BIRTH AND
                    01310 * NO CHANGE ROUTINES.
                    01320 *
0CE2 8E   0000      01330         LDX     #$0000    * POINT X TO VIDEO DISPLAY
0CE5 108E 1000      01340         LDY     #$1000    * POINT Y TO SCRATCHPAD
                    01350 *
0CE9 C6   04        01360 CIRCLE  LDB     #$04      * COUNT FOUR QUARTER BYTES
0CEB A6   84        01370         LDA     ,X        * GET VIDEO DISPLAY BYTE
0CED 34   10        01380         PSHS    X         * STASH X REGISTER FOR LATER
0CEF 34   03        01390         PSHS    A,CC      * STASH VIDEO, CARRY INFO
0CF1 35   03        01400 HERE    PULS    A,CC      * RESTORE VIDEO, CARRY INFO
0CF3 49             01410         ROLA              * ROTATE THROUGH CARRY FLAG
0CF4 49             01420         ROLA              * ROTATE THROUGH CARRY FLAG
0CF5 34   03        01430         PSHS    A,CC      * RE-SAVE ROTATING A, CARRY
0CF7 A6   A0        01440         LDA     ,Y+       * GET VALUE FROM SCRATCHPAD
```

```
0CF9 48          01450          ASLA              * DOUBLE IT (2-BYTE OFFSET)
0CFA 8E  0CFF    01460          LDX     #ZAP      * GET START OF TABLE
0CFD 6E  96      01470          JMP     ↑A,X←     * ADD OFFSET & JUMP TO ROUTINE
                 01480 *
0CFF     0D11    01490 ZAP      FDB     DEATH     * ROUTINE NEIGHBORHOOD = 0
0D01     0D11    01500          FDB     DEATH     * ROUTINE NEIGHBORHOOD = 1
0D03     0D19    01510          FDB     NOCHNG    * ROUTINE NEIGHBORHOOD = 2
0D05     0D25    01520          FDB     BIRTH     * ROUTINE NEIGHBORHOOD = 3
0D07     0D11    01530          FDB     DEATH     * ROUTINE NEIGHBORHOOD = 4
0D09     0D11    01540          FDB     DEATH     * ROUTINE NEIGHBORHOOD = 5
0D0B     0D11    01550          FDB     DEATH     * ROUTINE NEIGHBORHOOD = 6
0D0D     0D11    01560          FDB     DEATH     * ROUTINE NEIGHBORHOOD = 7
0D0F     0D11    01570          FDB     DEATH     * ROUTINE NEIGHBORHOOD = 8
                 01580 *
                 01590 * THE DEATH ROUTINE MUST CREATE COLOR VALUE 00 ON THE
                 01600 * COLOR GRAPHICS DISPLAY SCREEN.  HALF OF THIS VALUE
                 01610 * IS PRESENTLY IN THE A ACCUMULATOR, AND THE OTHER
                 01620 * HALF IS IN THE CARRY FLAG.  BOTH ARE SET TO ZERO
                 01630 * IN THIS ROUTINE.
                 01640 *
0D11 35  03      01650 DEATH    PULS    A,CC      * SAVE DISPLAY, ROTATING BIT
0D13 84  FE      01660          ANDA    #$FE      * MASK OUT BIT ZERO
0D15 1C  FE      01670          ANDCC   #$FE      * MASK OUT CARRY BIT
0D17 20  18      01680          BRA     OUT       * GO OUT TO ROTATE & DISPLAY
                 01690 *
                 01700 * THE NO-CHANGE ROUTINE IS NOT PRECISELY THAT IN THIS
                 01710 * CASE.  COLORS IN THIS GRAPHICS MODE ARE BUFF, CYAN,
                 01720 * MAGENTA AND ORANGE, AS REPRESENTED BY PATTERNS 00,
                 01730 * 01, 10 AND 11.  IN THIS PROGRAM, DORMANT CELLS ARE
                 01740 * SHOWN IN BUFF (00), NEWBORN CELLS IN CYAN (01), AND
                 01750 * CELLS OLDER THAN ONE GENERATION AS ORANGE (11).  THE
                 01760 * VALUE 10 (MAGENTA) IS DEFINED AS ILLEGAL.  HOWEVER,
                 01770 * SHOULD IT OCCUR IN THE "GARDEN OF EDEN", IT MUST BE
                 01780 * CHANGED TO 00.  THIS ROUTINE MAKES THE CHANGE.
                 01790 *
0D19 35  03      01800 NOCHNG   PULS    A,CC      * RESTORE ROTATED DISPLAY INFO
0D1B 25  04      01810          BCS     HIGH      * IF SET, MEANS BIT ZERO = 1
0D1D 84  FE      01820          ANDA    #$FE      * MAKE 10 OR 00 BECOME 00
0D1F 20  10      01830          BRA     OUT       * AND GO OUT, STORE & DISPLAY
0D21 8A  01      01840 HIGH     ORA     #$01      * MAKE 01 OR 11 BECOME 11
0D23 20  0C      01850          BRA     OUT       * AND GO OUT, STORE & DISPLAY
                 01860 *
                 01870 * THE BIRTH ROUTINE IS ALSO A "NO CHANGE" ROUTINE IF
                 01880 * THE NUMBER OF NEIGHBORS IS PRECISELY THREE AND A
                 01890 * LIVE CELL ALREADY EXISTS.  VALUES FOR NEWBORNS ARE
                 01900 * GIVEN AS 01 (CYAN), BUT ALREADY EXISTING CELLS MUST
                 01910 * BE CHANGED TO OLDER CELLS IN ORANGE (11).  ALSO,
                 01920 * ANY ILLEGALS (10, MAGENTA) MUST BE CHANGED.
                 01930 *
0D25 35  03      01940 BIRTH    PULS    A,CC      * GET ROTATED DISPLAY VALUE
0D27 24  04      01950          BCC     LOW       * GO IF CARRY = 0 (00 OR 10)
0D29 8A  01      01960          ORA     #$01      * IF C=1, MAKE 11 = OLDSTER
0D2B 20  04      01970          BRA     OUT       * GO OUT, STORE AND DISPLAY
0D2D 84  FE      01980 LOW      ANDA    #$FE      * C = 0; MAKE 00 OR 01 BE 00
0D2F 1A  01      01990          ORCC    #$01      * THEN MAKE VALUE BECOME 01
                 02000 *
                 02010 * THE "OUT" ROUTINE IS AN ORDERLY EXIT, TESTING FOR
                 02020 * THE ROTATED POSITION OF A (THE QUARTER-BYTE COUNT),
                 02030 * DOING THE FINAL (NINTH) ROTATE TO GET THE BYTE
                 02040 * BACK IN POSITION IF NECESSARY, STORING THE FINAL
                 02050 * BYTE IN A TEMPORARY SCREEN, AND BRANCHING BACK IF
                 02060 * THE ENTIRE 3,072 BYTE BLOCK (12,288 CELLS) HAS NOT
                 02070 * BEEN DONE.
                 02080 *
0D31 34  03      02090 OUT      PSHS    A,CC      * STASH ROTATING BIT, VIDEO
0D33 5A          02100          DECB              * TEST FOR NEXT QUARTER BYTE
0D34 26  BB      02110          BNE     HERE      * IF NOT DONE, GET NEXT QUARTER
0D36 35  03      02120          PULS    A,CC      * RESTORE ROTATING BIT, VIDEO
0D38 35  10      02130          PULS    X         * RECOVER STASHED X REGISTER
0D3A 49          02140          ROLA              * ROTATE TO RESTORE POSITION
0D3B A7  89 1000 02150          STA     $1000,X   * AND STORE BACK INTO DISPLAY
0D3F 30  01      02160          LEAX    1,X       * GET NEXT POSITION IN PLACE
0D41 8C  0C00    02170          CMPX    #$0C00    * SEE IF END OF DISPLAY YET
0D44 26  A3      02180          BNE     CIRCLE    * IF NOT, BACK FOR NEXT BYTE
                 02190 *
                 02200 * THE FOLLOWING ROUTINE REDIRECTS THE SCREEN TO $1000,
                 02210 * WHERE THE NEW GENERATION HAS BEEN CREATED.  IT THEN
                 02220 * COPIES THAT INFORMATION INTO THE SCREEN STARTING AT
                 02230 * $0000, AND SWITCHES SCREENS.  THIS WORK-AND-SWITCH
                 02240 * PROCESS PREVENTS THE NEW GENERATION FROM SWIMMING
                 02250 * DOWNWARD OVER THE PREVIOUS GENERATION AS YOU WATCH.
                 02260 *
0D46 B7  FFCD    02270          STA     $FFCD     * SWITCH TO SCREEN AT $1000
```

```
                        02280 *
0D49 8E   1000          02290           LDX     #$1000  * POINT X TO NEW SCREEN
0D4C 108E 0000          02300           LDY     #$0000  * POINT Y TO OLD SCREEN
0D50 A6   80            02310 XFER      LDA     ,X+     * GET VALUE FROM NEW SCREEN
0D52 A7   A0            02320           STA     ,Y+     * TRANSFER VALUE TO OLD SCREEN
0D54 8C   1C00          02330           CMPX    #$1C00  * SEE IF SCREEN IS FINISHED
0D57 26   F7            02340           BNE     XFER    * GO BACK TO FINISH SCREEN
                        02350 *
0D59 B7   FFCC          02360           STA     $FFCC   * REDIRECT VIDEO TO $0000
                        02370 *
0D5C 7E   0C9A          02380           JMP     AGAIN   * AND REPEAT THE WHOLE PROCESS
                        02390 *
          0C80          02400           END     START
00000 TOTAL ERRORS
AGAIN   0C9A
BIRTH   0D25
CIRCLE  0CE9
DEATH   0D11
HERE    0CF1
HIGH    0D21
LOW     0D2D
NEXT1   0CA4
NEXTQ   0CD8
NOCHNG  0D19
NXTCEL  0CB7
OUT     0D31
QUARTR  0CBB
START   0C80
VIDEO   0C8C
XFER    0D50
ZAP     0CFF
```

* In this video display, if the A accumulator contains 11110100, what cells are present?

Mature, mature, newborn and dormant.

* If the A accumulator contains 10110000, what cells are present?

Illegal, mature, dormant and dormant.

* What generation is this? Why?

The Garden of Eden, because illegal cells cannot occur in subsequent generations to the Garden of Eden.

* A contains 10110000 in the Garden of Eden. If the algorithm says all cells remain unchanged -- in terms of this Game of Life -- what will the A accumulator contain in the next generation? Why?

A will contain 01110000 because illegals are changed to newborns after the Garden of Eden.

Program #27B, an object code program. Turn on the power to your Extended Color BASIC computer. When the cursor appears, type CLOAD and press ENTER. The computer will search (S) and find (F). When the cursor reappears, type EXEC and press ENTER. The program will execute automatically, If an I/O error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

Watching the Game of Life is a fascinating experience. A lot has been written about this pastime, and versions in three dimensions and many colors have been developed.

My intention with these three lessons was not only to introduce the concept of indirect indexed addressing, but also to demonstrate with an apparently complicated example the idea of compartmentalized or modular programming. The modules were designed for speed, but with little modification they could be used as complete subroutines . . . those that select color sets, video display memory, and graphics modes and the one to fill memory are complete. The Life routines consist of the evaluation block and the more complex regeneration section. I've drawn arrows in the program listing to show the clear program flow. I'll also tell you that this program wasn't an off-the-cuff creation; it was in fact revised nearly 20 times before it was ready for you to see. Not that it didn't work until 20 tries, but rather that I used more instructions than I needed to do some of the work. In looking for economies of speed, I was able to rethink the details of each routine. You'll do

that too as you attempt larger-scale programs.

Here's a summary of the concepts that you have seen:

1. You should be able to establish video modes by referring to the SAM and VDG setup charts and the Color Computer memory map.

2. You should be able to set A, B, X and Y registers and then use the push stack instructions for fast memory filling or clearing.

3. You shold be able to set the stack pointer to a specific place in memory using the LDS instruction.

4. You should understand how to use the rotate instruction to rotate part of a byte into the carry flag, and then employ that information for program branching.

5. You should be able to use a fixed pointer plus a variable offset to select an address from a table of addresses, and then access the information at the resulting address. In other words, you should understand indirect indexed addressing.

6. You should be able to directly manipulate the condition codes (in this example, the carry flags and interrupt masks) using ANDCC and ORCC instructions.

7. You should understand the whys and hows of switching video display modes to hide manipulation of memory.

8. You should have read about pseudo-ops in the EDTASM+ manual, and be able to use ORG, EQU, END, FCB, FCC, FDB and RMB. In summary, these are:

ORG defines the first byte of the program.

EQU identifies the value of a label.

END concludes the assembly process.

FCB identifies a specific byte to be placed in memory.

FDB identifies a spedific two-byte word to be placed in memory.

FCC identifies an ASCII string to be placed in memory.

RMB tells the assembler to reserve — that is, not to use — a block of memory.

I hope all these concepts are clear to you. If you don't feel completely confident, please review. Review the written text for specific items, and review all three lessons if you don't think you could create a complete body of assembly

```
* If A is 01110000, what are the values of A in binary and hexadecimal, and the value of the carry flag, when ROLA is executed four times? (Assume the carry flag is zero to start).

Start A = 01110000, 70; C=0
      A = 11000000, C0; C=1
      A = 00000011, 03; C=1
      A = 00001110, 0E; C=0
      A = 00111000, 38; C=0

* The scratchpad memory in this Game of Life is used for two purposes; what are they?

1. To store the neighborhood count during evaluation.
2.  To build a new screen containing the next generation.

* What is an algorithm?

A general term for a set of rules.

* What is a pixel?

A picture element.

* What does VDG mean?

Video Display Generator.

* How many pixels does the Color Computer's VDG provide?

256 horizontal by 192 vertical pixels.

* How many different points are displayed on the screen in the most detailed graphics mode (mode RG6)?

49,152 points.

* How many bytes are required for the most detailed graphics mode (mode RG6)?

6,144 bytes.
```

Summary

* Why can 6,144 bytes display 49,152 points?

Because one byte represents eight display points.

* What addressing mode is JMP $3456?

Extended addressing.

* What addressing mode is JMP A,X?

Indexed addressing.

* What addressing mode is JMP [A,X]?

Indexed indirect addressing.

* What does the instruction LDS #$1000 do?

It loads the hardware stack pointer (S) with the value $1000.

* What does the instruction ORCC #$50 do?

It turns off the interrupts.

* What kind of instruction is FDB $A0D7?

It is an assembler pseudo-op.

* What does FDB $A0D7 do?

It tells the assembler to place the two-byte word $A0D7 in memory.

* What is an addressing mode?

The way in which a machine language instruction gets its information.

code to solve a similar programming problem. These three lessons have offered approach, conceptualization, decision-making, and programming technique. These three lessons — in fact, the past *five* lessons — are the gateway to the rest of this course. I urge you to understand them well. Till next time.