

Chapter 1

Code Runner

1.1 Introduction

For `dasm`, we'll first specify that we're targeting the 6502 processor:

```
1   $\langle preamble\ 1 \rangle \equiv$  (16)  
    PROCESSOR 6502
```

That is all!

Chapter 2

Apple II Graphics

Hi-res graphics on the Apple II is odd. Graphics are memory-mapped, not exactly consecutively, and bits don't always correspond to pixels. Color especially is odd, compared to today's luxurious 32-bit per pixel RGBA.

2.1 Pixels and their color

First we'll talk about pixels. Nominally, the resolution of the hi-res graphics screen is 280 pixels wide by 192 pixels tall. In the memory map, each row is represented by 40 bytes. The high bit of each byte is not used for pixel data, but is used to control color.

Here are some rules for how these bytes are turned into pixels:

- Pixels are drawn to the screen from byte data least significant bit first. This means that for the first byte bit 0 is column 0, bit 1 is column 1, and so on.
- A pattern of 11 results in two white pixels at the 1 positions.
- A pattern of 010 results at least in a colored pixel at the 1 position.
- A pattern of 101 results at least in a colored pixel at the 0 position.
- So, a pattern of 01010 results in at least three consecutive colored pixels starting from the first 1 to the last 1. The last 0 bit would also be colored if followed by a 1.
- Likewise, a pattern of 11011 results in two white pixels, a colored pixel, and then two more white pixels.
- The color of a 010 pixel depends on the column that the 1 falls on, and also whether the high bit of its byte was set or not.
- The color of a 11011 pixel depends on the column that the 0 falls on, and also whether the high bit of its byte was set or not.

	Odd	Even
High bit clear	Green	Violet
High bit set	Orange	Blue

The implication is that you can only select one pair of colors per byte.

An example would probably be good here. We will take one of the sprites from the game.

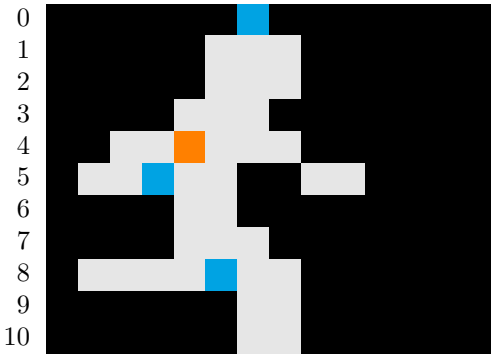
Bytes	Bits		Pixel Data
00 00	0000000	0000000	00000000000000
00 00	0000000	0000000	00000000000000
00 00	0000000	0000000	00000000000000
55 00	1010101	0000000	10101010000000
41 00	1000001	0000000	10000010000000
01 00	0000001	0000000	10000000000000
55 00	1010101	0000000	10101010000000
50 00	1010000	0000000	00001010000000
50 00	1010000	0000000	00001010000000
51 00	1010001	0000000	10001010000000
55 00	1010101	0000000	10101010000000

The game automatically sets the high bit of each byte, so we know we're going to see orange and blue. Assuming that the following bits are all zero, and we place the sprite starting at column 0, we should see this:



Here is a more complex sprite:

Bytes		Bits		Pixel Data
40	00	1000000	0000000	00000010000000
60	01	1100000	0000001	00000111000000
60	01	1100000	0000001	00000111000000
70	00	1110000	0000000	00001110000000
6C	01	1101100	0000001	00110111000000
36	06	0110110	0000110	01101100110000
30	00	0110000	0000000	00001100000000
70	00	1110000	0000000	00001110000000
5E	01	1011110	0000001	01111011000000
40	01	1000000	0000001	00000011000000
40	01	1000000	0000001	00000011000000



Take note of the orange and blue pixels. All the patterns noted in the rules above are used.

2.2 The sprites

Lode Runner defines 104 sprites, each being 11 rows, with two bytes per row. The first bytes of all 104 sprites are in the table first, then the second bytes, then the third bytes, and so on. Later we will see that only the leftmost 10 pixels out of the 14-pixel description is used.

4<tables 4>≡6>

ORG\$AD00

SPRITE_DATA:

INCLUDE "sprite_data.asm"

Defines:

SPRITE_DATA, used in chunk 8.

2.3 Shifting sprites

This is all very good if we're going to draw sprites exactly on 7-pixel boundaries, but what if we want to draw them starting at other columns? In general, such a shifted sprite would straddle three bytes, and Lode Runner sets aside an area of memory at the end of zero page for 11 rows of three bytes that we'll write to when we want to compute the data for a shifted sprite.

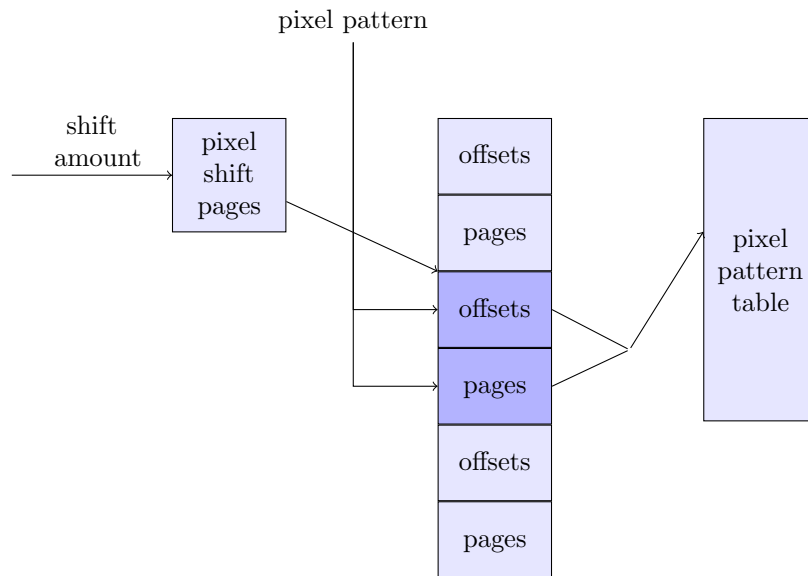
```
5  <defines 5>≡ 7c>
    ORG      $DF
    BLOCK_DATA      DS      33
```

Defines:
BLOCK_DATA, used in chunks 8 and 14.

Code Runner also contains tables which show how to shift any arbitrary 7-pixel pattern right by any amount from zero to six pixels.

For example, suppose we start with a pixel pattern of 0110001, and we want to shift that right by three bits. The 14-bit result would be 0000110 0010000. However, we have to break that up into bytes, reverse the bits (remember that each byte's bits are output as pixels least significant bit first), and set their high bits, so we end up with 10110000 10000100.

Now, given a shift amount and a pixel pattern, we should be able to find the two-byte shifted pattern. Code Runner accomplishes this with table lookups as follows:



The pixel pattern table is a table of every possible pattern of 7 consecutive pixels spread out over two bytes. This table is 512 entries, each entry being two bytes. A naive table would have redundancy. For example the pattern 0000100 starting at column 0 is exactly the same as the pattern 0001000 starting at column 1. This table eliminates that redundancy.

```
6 <tables 4>+≡ <4 7a>
    ORG      $A900
    PIXEL_PATTERN_TABLE:
    INCLUDE "pixel_pattern_table.asm"

Defines:
    PIXEL_PATTERN_TABLE, never used.
```

Now we just need tables which index into `PIXEL_PATTERN_TABLE` for every 7-pixel pattern and shift value. This table works by having the page number for the shifted pixel pattern at index `shift * 0x100 + 0x80 + pattern` and the offset at index `shift * 0x100 + pattern`.

```
7a  <tables 4>+≡                                     <6 7b>
      ORG      $A200
      PIXEL_SHIFT_TABLE:
      INCLUDE "pixel_shift_table.asm"
```

Defines:
`PIXEL_SHIFT_TABLE`, never used.

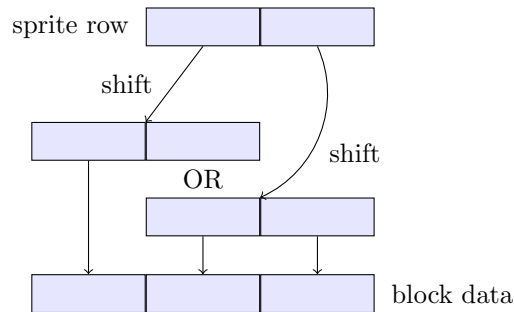
Rather than multiplying the shift value by `0x100`, we instead define another table which holds the page numbers for the shift tables for each shift value.

```
7b  <tables 4>+≡                                     <7a 10a>
      ORG      $84C1
      PIXEL_SHIFT_PAGES:
      HEX      A2 A3 A4 A5 A6 A7 A8
```

Defines:
`PIXEL_SHIFT_PAGES`, used in chunk 8.

So we can get shifted pixels by indexing into all these tables.

Now we can define a routine that will take a sprite number and a pixel shift amount, and write the shifted pixel data into the `BLOCK_DATA` area. The routine first shifts the first byte of the sprite into a two-byte area. Then it shifts the second byte of the sprite, and combines that two-byte result with the first. Thus, we shift two bytes of sprite data into a three-byte result.



Rather than load addresses from the tables and store them, the routine modifies its own instructions with those addresses.

```
7c  <defines 5>+≡                                     <5 10b>
      ORG      $0A
      TMP_PTR   DS.W    1
      ORG      $1D
      ROW_COUNT DS      1
      SPRITE_NUM DS      1
```

Defines:
`ROW_COUNT`, used in chunks 8 and 14.
`SPRITE_NUM`, used in chunks 8 and 14.
`TMP_PTR`, used in chunk 8.

```

8      (routines 8)≡
      ORG      $8438
      COMPUTE_SHIFTED_SPRITE:
      SUBROUTINE
      ; Enter routine with X set to pixel shift amount and
      ; SPRITE_NUM containing the sprite number to read.

      .offset_table      EQU $A000      ; Target addresses in read
      .page_table        EQU $A080      ; instructions. The only truly
      .shift_ptr_byte0    EQU $A000      ; necessary value here is the
      .shift_ptr_byte1    EQU $A000      ; 0x80 in .shift_ptr_byte0.

      LDA      #$0B      ; 11 rows
      STA      ROW_COUNT
      LDA      #<SPRITE_DATA
      STA      TMP_PTR
      LDA      #>SPRITE_DATA
      STA      TMP_PTR+1      ; TMP_PTR = SPRITE_DATA
      LDA      PIXEL_SHIFT_PAGES, X
      STA      .rd_offset_table + 2
      STA      .rd_page_table + 2
      STA      .rd_offset_table2 + 2
      STA      .rd_page_table2 + 2      ; Fix up pages in lookup instructions
      ; based on shift amount (X).

      LDX      #$00      ; X is the offset into BLOCK_DATA.

      .loop:      ; === LOOP === (over all 11 rows)
      LDY      SPRITE_NUM
      LDA      (TMP_PTR), Y
      TAY      ; Get sprite pixel data.

      .rd_offset_table:
      LDA      .offset_table, Y      ; Load offset for shift amount.
      STA      .rd_shift_ptr_byte0 + 1
      CLC
      ADC      #$01
      STA      .rd_shift_ptr_byte1 + 1      ; Fix up instruction offsets with it.
      .rd_page_table:
      LDA      .page_table, Y      ; Load page for shift amount.
      STA      .rd_shift_ptr_byte0 + 2
      STA      .rd_shift_ptr_byte1 + 2      ; Fix up instruction page with it.

      .rd_shift_ptr_byte0:
      LDA      .shift_ptr_byte0      ; Read shifted pixel data byte 0
      STA      BLOCK_DATA, X      ; and store in block data byte 0.
      .rd_shift_ptr_byte1:
      LDA      .shift_ptr_byte1      ; Read shifted pixel data byte 1
      STA      BLOCK_DATA+1, X      ; and store in block data byte 1.

```



```

    LDA    TMP_PTR
    CLC
    ADC    #$68
    STA    TMP_PTR
    LDA    TMP_PTR+1
    ADC    #$00
    STA    TMP_PTR+1                ; TMP_PTR++

    ; Now basically do the same thing with the second sprite byte

    LDY    SPRITE_NUM
    LDA    (TMP_PTR), Y
    TAY                                ; Get sprite pixel data.

.rd_offset_table2:
    LDA    .offset_table, Y          ; Load offset for shift amount.
    STA    .rd_shift_ptr2_byte0 + 1
    CLC
    ADC    #$01
    STA    .rd_shift_ptr2_byte1 + 1  ; Fix up instruction offsets with it.
.rd_page_table2:
    LDA    .page_table, Y           ; Load page for shift amount.
    STA    .rd_shift_ptr2_byte0 + 2
    STA    .rd_shift_ptr2_byte1 + 2 ; Fix up instruction page with it.

.rd_shift_ptr2_byte0:
    LDA    .shift_ptr_byte0         ; Read shifted pixel data byte 0
    ORA    BLOCK_DATA+1, X          ; OR with previous block data byte 1
    STA    BLOCK_DATA+1, X          ; and store in block data byte 1.
.rd_shift_ptr2_byte1:
    LDA    .shift_ptr_byte1         ; Read shifted pixel data byte 1
    STA    BLOCK_DATA+2, X          ; and store in block data byte 2.

    LDA    TMP_PTR
    CLC
    ADC    #$68
    STA    TMP_PTR
    LDA    TMP_PTR+1
    ADC    #$00
    STA    TMP_PTR+1                ; TMP_PTR++

    INX
    INX
    INX                                ; X += 3
    DEC    ROW_COUNT                ; ROW_COUNT--
    BNE    .loop                    ; loop while ROW_COUNT > 0
    RTS

```

Defines:

COMPUTE_SHIFTED_SPRITE, used in chunk 14.

Uses BLOCK_DATA 5, PIXEL_SHIFT_PAGES 7b, ROW_COUNT 7c, SPRITE_DATA 4, SPRITE_NUM 7c,

and TMP_PTR 7c.

2.4 Memory mapped graphics

The Apple II maps the area from \$2000-\$3FFF to high-res graphics page 1 (HGR1), and \$4000-\$5FFF to page 2 (HGR2). Within a row, consecutive bytes map to consecutive pixels. However, rows themselves are not consecutive in memory.

To make it easy to convert a row number from 0 to 191 to a base address, Lode Runner has a table and a routine to use that table.

```

10a  <tables 4>+≡                                     <7b 11>
      ORG      $1A85
      ROW_TO_OFFSET_LO:
      INCLUDE "row_to_offset_lo_table.asm"
      ROW_TO_OFFSET_HI:
      INCLUDE "row_to_offset_hi_table.asm"

Defines:
      ROW_TO_OFFSET_HI, used in chunk 10c.
      ROW_TO_OFFSET_LO, used in chunk 10c.

10b  <defines 5>+≡                                     <7c 13a>
      ORG      $0C
      ROW_ADDR  DS      2
      ORG      $1F
      HGR_PAGE  DS      1          ; 0x20 for HGR1, 0x40 for HGR2

Defines:
      HGR_PAGE, used in chunks 10c and 14.
      ROW_ADDR, used in chunks 10c and 14.

10c  <routines 8>+≡                                     <8 12>
      ORG      $7A31
      ROW_TO_ADDR:
      SUBROUTINE
      ; Enter routine with Y set to row. Base address
      ; (for column 0) will be placed in ROW_ADDR.

      LDA      ROW_TO_OFFSET_LO, Y
      STA      ROW_ADDR
      LDA      ROW_TO_OFFSET_HI, Y
      ORA      HGR_PAGE
      STA      ROW_ADDR+1
      RTS

Defines:
      ROW_TO_ADDR, used in chunk 14.
Uses HGR_PAGE 10b, ROW_ADDR 10b, ROW_TO_OFFSET_HI 10a, and ROW_TO_OFFSET_LO 10a.

```

Lode Runner's screens are organized into 28 sprites across by 17 sprites down. To convert between sprite coordinates and screen coordinates, we use tables and lookup routines. Each sprite is 10 pixels across by 11 pixels down.

```

11  <tables 4>+≡<10a 13b>
      ORG      $1C35
      ROW_TABLE2:
          ; 28 rows of 5 pixels each
      HEX      00 05 0a 0f 14 19 1e 23 28 2d 32 37 3c 41 46 4b
      HEX      50 55 5a 5f 64 69 6e 73 78 7d 82 87
      ROW_TABLE:
          ; 17 rows of 11 pixels each
      HEX      00 0B 16 21 2C 37 42 4D 58 63 6E 79 84 8F 9A A5
      HEX      B5
      COL_TABLE:
          ; Byte number
      HEX      00 01 02 04 05 07 08 0A 0B 0C 0E 0F 11 12 14 15
      HEX      16 18 19 1B 1C 1E 1F 20 22 23 25 26
      COL_SHIFT_TABLE:
          ; Right shift amount
      HEX      00 03 06 02 05 01 04 00 03 06 02 05 01 04 00 03
      HEX      06 02 05 01 04 00 03 06 02 05 01 04

```

Defines:

```

COL_SHIFT_TABLE, used in chunks 12 and 14.
COL_TABLE, used in chunks 12 and 14.
ROW_TABLE, used in chunks 12 and 14.
ROW_TABLE2, used in chunk 12.

```

12 $\langle routines\ 8 \rangle + \equiv$ $\langle 10c\ 14 \rangle$

```

    ORG      $885D
GET_ROWNUM_FOR:
    SUBROUTINE
    ; Enter routine with Y set to sprite row. On
    ; return, Y will be set to screen row.
    ; We can also set X to something, and on return
    ; X is set to something based on ROW_TABLE2, but
    ; so far I'm not sure what it's used for.

    LDA      ROW_TABLE, Y
    PHA
    LDA      ROW_TABLE2, X
    TAX                      ; X = ROW_TABLE2[X]
    PLA
    TAY                      ; Y = ROW_TABLE[Y]
    RTS

GET_COLNUM_FOR:
    SUBROUTINE
    ; Enter routine with X set to sprite number. On
    ; return, A will be set to screen column byte number
    ; and X will be set to an additional right shift amount.

    LDA      COL_TABLE, X
    PHA                      ; A = COL_TABLE2[X]
    LDA      COL_SHIFT_TABLE, X
    TAX                      ; X = COL_SHIFT_TABLE[X]
    PLA
    RTS

```

Defines:

GET_COLNUM_FOR, used in chunk 14.

GET_ROWNUM_FOR, used in chunk 14.

Uses COL_SHIFT_TABLE 11, COL_TABLE 11, ROW_TABLE 11, and ROW_TABLE2 11.

Now we can finally write the routines that draw a sprite on the screen. There are two entry points, one to draw on HGR1, and one for HGR2.

13a $\langle \text{defines } 5 \rangle + \equiv$ $\triangleleft 10b$

```

    ORG      $1B
    ROWNUM          DS      1
    COLNUM          DS      1
    ORG      $50
    MASK0           DS      1
    MASK1           DS      1
    ORG      $71
    COL_SHIFT_AMT   DS      1
    ORG      $85
    GAME_COLNUM     DS      1
    GAME_ROWNUM     DS      1

```

Defines:

```

COL_SHIFT_AMT, used in chunk 14.
COLNUM, used in chunk 14.
GAME_COLNUM, used in chunk 14.
GAME_ROWNUM, used in chunk 14.
ROWNUM, used in chunk 14.

```

13b $\langle \text{tables } 4 \rangle + \equiv$ $\triangleleft 11$

```

    ORG      $8328
    PIXEL_MASK0:
    BYTE      %00000000
    BYTE      %00000001
    BYTE      %00000011
    BYTE      %00000111
    BYTE      %00001111
    BYTE      %00011111
    BYTE      %00111111
    PIXEL_MASK1:
    BYTE      %11111000
    BYTE      %11110000
    BYTE      %11100000
    BYTE      %11000000
    BYTE      %10000000
    BYTE      %11111110
    BYTE      %11111100

```

Defines:

```

PIXEL_MASK0, used in chunk 14.
PIXEL_MASK1, used in chunk 14.

```

```

14  < routines 8 > +≡ < 12
    ORG      $82AA
DRAW_SPRITE_PAGE1:
    SUBROUTINE
    ; Enter routine with A set to sprite number to draw,
    ; GAME_ROWNUM set to the row to draw it at, and GAME_COLNUM
    ; set to the column to draw it at.

    STA      SPRITE_NUM
    LDA      #$20                ; Page number for HGR1
    BNE      DRAW_SPRITE        ; Actually unconditional jump

DRAW_SPRITE_PAGE2:
    SUBROUTINE
    ; Enter routine with A set to sprite number to draw,
    ; GAME_ROWNUM set to the row to draw it at, and GAME_COLNUM
    ; set to the column to draw it at.

    STA      SPRITE_NUM
    LDA      #$40                ; Page number for HGR2
    ; fallthrough

DRAW_SPRITE:
    STA      HGR_PAGE
    LDY      GAME_ROWNUM
    JSR      GET_ROWNUM_FOR
    STY      ROWNUM              ; ROWNUM = ROW_TABLE[GAME_ROWNUM]

    LDX      GAME_COLNUM
    JSR      GET_COLNUM_FOR
    STA      COLNUM              ; COLNUM = COL_TABLE[GAME_COLNUM]
    STX      COL_SHIFT_AMT       ; COL_SHIFT_AMT = COL_SHIFT_TABLE[GAME_COLNUM]

    LDA      PIXEL_MASK0, X
    STA      MASK0               ; MASK0 = PIXEL_MASK0[COL_SHIFT_AMT]
    LDA      PIXEL_MASK1, X
    STA      MASK1               ; MASK1 = PIXEL_MASK1[COL_SHIFT_AMT]

    JSR      COMPUTE_SHIFTED_SPRITE

    LDA      #$0B
    STA      ROW_COUNT
    LDX      #$00
    LDA      COL_SHIFT_AMT
    CMP      #$05
    BCS      .need_3_bytes        ; if COL_SHIFT_AMT >= 5, we need to alter three screen bytes.

.loop1:
    LDY      ROWNUM
    JSR      ROW_TO_ADDR

```

```

LDY    COLNUM
LDA     (ROW_ADDR), Y
AND     MASK0
ORA     BLOCK_DATA, X
STA     (ROW_ADDR), Y      ; screen[COLNUM] = screen[COLNUM] & MASK0 | BLOCK_DATA[X]

INX
INX     ; X++
INY     ; Y++
LDA     (ROW_ADDR), Y
AND     MASK1
ORA     BLOCK_DATA+1, X
STA     (ROW_ADDR), Y      ; screen[COLNUM+1] = screen[COLNUM+1] & MASK1 | BLOCK_DATA[X+1]

INX
INX     ; X += 2
INC     ROWNUM              ; ROWNUM++
DEC     ROW_COUNT           ; ROW_COUNT--
BNE     .loop1              ; loop while ROW_COUNT > 0
RTS

```

.need_3_bytes

Defines:

DRAW.Sprite.PAGE1, never used.

DRAW.Sprite.PAGE2, never used.

Uses BLOCK_DATA 5, COL_SHIFT_AMT 13a, COL_SHIFT_TABLE 11, COL_TABLE 11, COLNUM 13a, COMPUTE_SHIFTED_SPRITE 8, GAME_COLNUM 13a, GAME_ROWNUM 13a, GET_COLNUM_FOR 12, GET_ROWNUM_FOR 12, HGR_PAGE 10b, PIXEL_MASK0 13b, PIXEL_MASK1 13b, ROW_ADDR 10b, ROW_COUNT 7c, ROW_TABLE 11, ROW_TO_ADDR 10c, ROWNUM 13a, and SPRITE_NUM 7c.

Chapter 3

The whole thing

We then put together the entire assembly file:

16 $\langle * 16 \rangle \equiv$
 $\langle preamble\ 1 \rangle$

Chapter 4

Defined Chunks

$\langle * 16 \rangle$ 16
 $\langle defines\ 5 \rangle$ 5, 7c, 10b, 13a
 $\langle preamble\ 1 \rangle$ 1, 16
 $\langle routines\ 8 \rangle$ 8, 10c, 12, 14
 $\langle tables\ 4 \rangle$ 4, 6, 7a, 7b, 10a, 11, 13b

Chapter 5

Index

BLOCK_DATA: 5, 8, 14
COL_SHIFT_AMT: 13a, 14
COL_SHIFT_TABLE: 11, 12, 14
COL_TABLE: 11, 12, 14
COLNUM: 13a, 14
COMPUTE_SHIFTED_SPRITE: 8, 14
DRAW_SPRITE_PAGE1: 14
DRAW_SPRITE_PAGE2: 14
GAME_COLNUM: 13a, 14
GAME_ROWNUM: 13a, 14
GET_COLNUM_FOR: 12, 14
GET_ROWNUM_FOR: 12, 14
HGR_PAGE: 10b, 10c, 14
PIXEL_MASK0: 13b, 14
PIXEL_MASK1: 13b, 14
PIXEL_PATTERN_TABLE: 6
PIXEL_SHIFT_PAGES: 7b, 8
PIXEL_SHIFT_TABLE: 7a
ROW_ADDR: 10b, 10c, 14
ROW_COUNT: 7c, 8, 14
ROW_TABLE: 11, 12, 14
ROW_TABLE2: 11, 12
ROW_TO_ADDR: 10c, 14
ROW_TO_OFFSET_HI: 10a, 10c
ROW_TO_OFFSET_LO: 10a, 10c
ROWNUM: 13a, 14
SPRITE_DATA: 4, 8
SPRITE_NUM: 7c, 8, 14
TMP_PTR: 7c, 8