PPU sprite evaluation

From Nesdev wiki

PPU sprite evaluation is an operation done by the PPU once each scanline. It prepares the set of sprites and fetches their data to be rendered on the next scanline.

This is a separate step from sprite rendering.

Contents

- 1 Overview
- 2 Details
- 3 Sprite overflow bug
  - 3.1 Cause of the sprite overflow bug
  - 3.2 Examples of usage
- 4 Notes
- 5 External links
- 6 References

Overview

Each scanline, the PPU reads the spritelist (that is, Object Attribute Memory) to see which to draw:

1. First, it clears the list of sprites to draw.
2. Second, it reads through OAM, checking which sprites will be on this scanline. It chooses the first eight it finds that do.
3. Third, if eight sprites were found, it checks (in a wrongly-implemented fashion) for further sprites on the scanline to see if the sprite overflow flag should be set.
4. Fourth, using the details for the eight (or fewer) sprites chosen, it determines which pixels each has on the scanline and where to draw them.

Details

During all visible scanlines, the PPU scans through OAM to determine which sprites to render on the next scanline. Sprites found to be within range are copied into the secondary OAM, which is then used to initialize eight internal sprite output units.

OAM[n][m] below refers to the byte at offset *4*n + m* within OAM, i.e. OAM byte *m* (0-3) of sprite *n* (0-63).

During each pixel clock (341 total per scanline), the PPU accesses OAM in the following pattern:

1. Cycles 1-64: Secondary OAM (32-byte buffer for current sprites on scanline) is initialized to $FF - attempting to read $2004 will return $FF. Internally, the clear operation is implemented by reading from the OAM and writing into the secondary OAM as usual, only a signal is active that makes the read always return $FF.
2. Cycles 65-256: Sprite evaluation

- *On odd cycles, data is read from (primary) OAM*
- *On even cycles, data is written to secondary OAM (unless secondary OAM is full, in which case it will read the value in secondary OAM instead)*
- 1. Starting at n = 0, read a sprite's Y-coordinate (OAM[n][0], copying it to the next open slot in secondary OAM (unless 8 sprites have been found, in which case the write is ignored).
    - 1a. If Y-coordinate is in range, copy remaining bytes of sprite data (OAM[n][1] thru OAM[n][3]) into secondary OAM.
- 2. Increment n
    - 2a. If n has overflowed back to zero (all 64 sprites evaluated), go to 4
    - 2b. If less than 8 sprites have been found, go to 1
    - 2c. If exactly 8 sprites have been found, disable writes to secondary OAM because it is full. This causes sprites in back to drop out.
- 3. Starting at m = 0, evaluate OAM[n][m] as a Y-coordinate.
    - 3a. If the value is in range, set the sprite overflow flag in $2002 and read the next 3 entries of OAM (incrementing 'm' after each byte and incrementing 'n' when 'm' overflows); if m = 3, increment n
    - 3b. If the value is not in range, increment n **and** m (without carry). If n overflows to 0, go to 4; otherwise go to 3
        - *The m increment is a hardware bug - if only n was incremented, the overflow flag would be set whenever more than 8 sprites were present on the same scanline, as expected.*
- 4. Attempt (and fail) to copy OAM[n][0] into the next free slot in secondary OAM, and increment n (repeat until HBLANK is reached)

3. Cycles 257-320: Sprite fetches (8 sprites total, 8 cycles per sprite)
    - 1-4: Read the Y-coordinate, tile number, attributes, and X-coordinate of the selected sprite from secondary OAM
    - 5-8: Read the X-coordinate of the selected sprite from secondary OAM 4 times (while the PPU fetches the sprite tile data)
    - For the first empty sprite slot, this will consist of sprite #63's Y-coordinate followed by 3 $FF bytes; for subsequent empty sprite slots, this will be four $FF bytes
4. Cycles 321-340+0: Background render pipeline initialization
    - Read the first byte in secondary OAM (while the PPU fetches the first two background tiles for the next scanline)

This pattern was determined by doing carefully timed reads from $2004 using various sets of sprites, and simulation in Visual 2C02 has subsequently confirmed this behavior.

Sprite overflow bug

During sprite evaluation, if eight in-range sprites have been found so far, the sprite evaluation logic continues to scan the primary OAM looking for one more in-range sprite to determine whether to set the sprite overflow flag. The first such check correctly checks the y coordinate of the next OAM entry, but after that the logic breaks and starts scanning OAM "diagonally", evaluating the tile number/attributes/X-coordinates of subsequent OAM entries as Y-coordinates (due to incorrectly incrementing m when moving to the next sprite). This results in inconsistent sprite overflow behavior showing both false positives and false negatives.

Cause of the sprite overflow bug

After investigation in Visual 2C02, the culprit of the sprite overflow bug appears to be the write disable signal that goes high after eight in-range sprites have been found (to prevent further updates to the secondary OAM), along with an error in the sprite evaluation logic.

As seen above, a side effect of the OAM write disable signal is to turn writes to the secondary OAM into reads from it. Once eight in-range sprites have been found, the value being read during write cycles from that point on is the y coordinate of the first sprite copied into the secondary OAM. Due to a logic error, the result of comparing this y coordinate to the current scanline number (which will always yield "in range", since the sprite would have had to be in range to get copied into the secondary OAM) is allowed to influence the sprite address incrementation logic, causing the glitchy updates to the sprite address seen above (due to how the timing works out). Once one more sprite has been found, another signal prevents the comparison from influencing the sprite address incrementation logic, and the bug is no longer in effect.

Examples of usage

For some examples of games using this bug/quirk, refer to the Sprite overflow games page.

Notes

- Sprite evaluation does not happen on the pre-render scanline. Because evaluation applies to the next line's sprite rendering, no sprites will be rendered on the first scanline, and this is why there is a 1 line offset on a sprite's Y coordinate.
- Sprite evaluation occurs if *either* the sprite layer or background layer is enabled via $2001. Unless both layers are disabled, it merely hides sprite rendering.
- Sprite evaluation does not cause sprite 0 hit. This is handled by sprite rendering instead.
- If the sprite address (OAMADDR, $2003) is not zero at the beginning of the pre-render scanline, on the 2C02 an OAM hardware refresh bug will cause the first 8 bytes of OAM to be overwritten by the 8 bytes beginning at OAMADDR & $F8 before sprite evaluation begins.[1][2]
- Visual 2C02 might be helpful when trying to understand how the algorithm operates and what the precise timings are.

External links

- Visual 2C02 logs of the PPU evaluating 1, 8, and 9 sprites (https://gist.github.com/beannaich/7a7ba066d909318debea) by beannaich

References

1. ↑ Forum (http://forums.nesdev.com/viewtopic.php?p=110019#p110019): Re: Just how cranky is the PPU OAM? (test notes by quietust)
2. ↑ Forum (http://forums.nesdev.com/viewtopic.php?f=3&t=12407): Huge Insect does not fully start