SD-8516 PPU

This is a short reference to the XY-2000 PPU.

Introduction

The following shows the speed tests and design decisions that went into the XY-2000 PPU.

Why a PPU? I'm not making one “just because a nintendo has one” or something like that. I'm making one because as it stands, the computer doesn't run fast enough to allow top-end game programming in BASIC.

Primitives

PPIXEL (plot_pixel)

Method	Pixels per Second
BASIC PIXEL command	2,100
INT 18h plot pixel	26,000
INT 18h plot pixel (no bounds check)	30,000
PPU via INT 0x18	90,000
PPU via INT 0x03 (direct)	220,000
PPIXEL (PPU via opcode)	460,000

The number shows the pixels per second of the screen being cleared in a tight loop. This essentially represents the fastest practical use for plot pixel; read X, Y and C data in a loop and plot it. If we unroll the loop by 20 times, the difference in speed between the INT 3 version and the OP.PPIXEL version remains about 2x, but you lose the loop overhead. The opcode version remains relatively 2x faster. This makes sense, since every call to INT 0x03 requires a LDA $0101 to access plot_pixel; the PPIXEL command gives this for free. Therefore PPIXEL is always at least twice as faster than INT 0x03 plot_pixel. It's actually a bit faster since it does not cross the host-guest bride inside the opcode call.

460,000 pixels/second is 7,600 pixels per frame at 60fps. That is almost exactly 30 16×16 sprites. Suggested use is to MEMCOPY a pre-drawn background into the frame-buffer and then draw sprites with PPIXEL (if you want to use PPIXEL for sprites). This way you don't have to draw the whole screen, and you don't have to draw the sprite background. It is hard to say how many sprites you will actually be able to draw in a frame because each sprite has more or less empty background space.

This initial test proves that a PPU construct has value, that it can serve as a drop-in boost to the code in INT 18h, that it should be initially implemented as a subcall of AH=$01, INT $03, and that the move to a dedicated opcode's practical value is 95% code density and 5% improved speed.

Code Replacement

PPIXEL can be called via INT 0x03:

```
  LDA  $0100       ; AH = 1 (PPU dispatch), AL = 00 (plot_pixel)
  INT  $03         ; plot_pixel(X, Y, C)
```

PPU plot_pixel replaces the old plot pixel function in the graphics library (function #1 in INT 18h). This is some of the oldest code in the entire system, likely carried over from the SD-8510/VC-2:

- [I:J] addressing mode
- Manual bank 2 register load (LDI #2)
- Register pressure PUSH X, PUSH C
- Overuse of LDA at start, then shows LDB before end
- Only uses A,B,X,Y,I,J,K,T

This was one of the first routines ever written for the original VC-2, as a test to draw characters to a terminal screen. This was at the time I was constructing the cpu emulator itself; during the writing of this function I added the LDB, PAB and UAB opcodes. That's a good reason to keep this code around, it serves as the design document for the framebuffer and renderer itself. In other words, “it works.” It's the source of truth over what it takes to plot a pixel in the framebuffer. The accelerated versions should get a unique entry inside INT 0x18, but not serve as drop-in replacements. That is probably the best way going forward.

```
;
================================================================================
; AH=01h - Plot MODE 3 Pixel (4bpp packed nibbles)
;
================================================================================
; Input:  X = x coordinate (mode dependant)
;         Y = y coordinate (mode dependant)
;         C = color (0-15)
; Output: CF = 0 on success, 1 if out of bounds
;
================================================================================
int18_plot_pixel_4bpp:
    PUSHA

    ; Bounds check using SCREEN_WIDTH / SCREEN_HEIGHT
    LDA [@SCREEN_WIDTH]
    CMP X, A
    JC @plot4_error
    LDA [@SCREEN_HEIGHT]
    CMP Y, A
    JC @plot4_error

    PUSH X                      ; save x
    PUSH C                      ; save color

    LDA [@SCREEN_WIDTH]
    SHR A                       ; A = stride (width / 2)
    MOV B, A                    ; B = stride
```

```
    MOV A, Y                    ; A = y
    MUL A, B                    ; A = y * stride

    POP C                       ; restore color
    POP B                       ; restore x into B
    PUSH B                      ; save x for nibble check
    MOV T, B
    SHR T
    ADD A, T                    ; A = byte offset

    MOV J, A
    LDI #2

    POP A                       ; A = x
    LDTL $01
    AND AL, TL
    CMP AL, $00
    JNZ @plot4_odd

plot4_even:
    LDBL [I:J]
    MOV AL, BL
    UAB                         ; AL = odd pixel, BL = even pixel
    MOV BL, CL                  ; BL = new color
    PAB                         ; AL = (new color << 4) | odd pixel
    MOV BL, AL
    STBL [I:J]
    POPA
    CLC
    RET

plot4_odd:
    LDBL [I:J]
    MOV AL, BL
    UAB                         ; AL = odd pixel, BL = even pixel
    MOV AL, CL                  ; AL = new color
    PAB                         ; AL = (even pixel << 4) | new color
    MOV BL, AL
    STBL [I:J]
    POPA
    CLC
    RET

plot4_error:
    POPA
    SEC
    RET
```

PLINE

These figures are for screen clears using LINE in a Y=0 to 199 loop.

Method	Pixels per Second
BASIC LINE command	22,000
INT 18h draw line	26,000
BASIC using PPU via INT 0x18	156,000
BASIC using PPU direct call	165,000
PPU via INT 0x18	15,000,000
PPU via INT 0x03 (direct)	55,000,000
PLINE (PPU via opcode)	77,000,000

As it turns out the line drawing algorithm, even though it is in assembly, is the limiting factor. The BASIC interpreter might execute 50 to 100 lines of assembly to get to INT 18h, but INT 18h is executing over 13,000 lines of assembly to draw a single horizontal line from 0,0 to 319,0. Thus replacing INT $18 with a PLINE opcode increased speed 7.5x to 165,000 pps fill rate. This implies 5-10 16×16 sprites with a smart draw algorithm. Of course, for BASIC, you have DRAWCHAR, and I suspect PBLIT and a series of SPRITE commands for BASIC will be the real breakthrough we need for our BASIC. But it's nice to know PLINE works well in BASIC. The numbers above represent 510 lines per second. With this much speed it is possible to write an ELITE clone in BASIC. That is going to be the benchmark.

The big win comes from the direct call in assembly. The speedup is equivalent to dropping five instructions; two INT calls, two RTI, and a LDA for the inner INT $3 call.

Fast tiles and sprites in BASIC with PLINE

If you have a smart drawing routine that reads contiguous pixels, then clusters of pixels can draw in ~1 cycle. So instead of having to call PPIXEL four times for the top row of a sprite, if the pixels are next to each other they can be detected as such as drawn via PLINE. While this might not be practical on-the-fly, it is interesting to consider a packed image format that would allow this to operate quickly; i.e. store the sprite data itself as a series of line draws. In a 16×16 sprite it is conceivable you could use about half the number of PLINE calls as you would use PPIXEL. The difference is that you would need a special format for storing the image data. This idea fits BASIC's “DATA” statements very well as it would be in a compressed format. The idea is you would store images in a kind of bytecode, and then the BASIC BLIT routine would read that format to quickly draw a sprite to screen. Of course, there is (will be) a PPU BLIT soon, so no one would use PLINE for this – but it's possible to do so.

PRECT and PFRECT

```
AH=$03 draw_rect
AH=$04 fill_rect
```
Tested and working. 160,000 rps and up in a normal loop.

PCIRCLE and PFCIRCLE

`AH=$05 draw_circle`
`AH=$06 fill_circle`
Draws circles. Tested and working. Circles draw depending on their size:

Radius	Draw speed	Example usage
10-20	150,000/sec	Games
80-100	18,000/sec	Art/Games
large	4500/sec	Art

For any meaningful use case, this is probably good enough. At 100,000+ circles a second at radius 30 and under, you could write any kind of game with this circle.

PCLEAR

`AH=$07 clear-to-color`
Clears screen to color in cl. Tested and working.

Sprites

Sprites are arguably the reason for a PPU, and why it's called a PPU and not a 2d accelerator.

- INT 18h LOAD and DRAW for 4bpp and 8bpp: Tested and working

The INT 18h DRAW function can draw a 16×16 sprite in 1.3ms. This is very slow.

```
spritebench_loop:
    ; Clear screen
    LDA  $0100              ; CMD_CLEAR
    LDCL $00
    INT  $03

    LDELM $A000
    LDFLD $020000
    CALL @draw_sprite_data

    INC X
    CMP X, #303
    JC @done

    LDAH $52        ; set VSTEP mode (step once)
    INT 0x18

spin_wait_loop:
    LDAH $53
```

```
    INT 0x18
    CMP AL, 1
    JNZ @spin_wait_loop

    JMP @spritebench_loop
```

If you take out the spin-wait loop, the clear screen and the VSTEP instruction, this code executes 303 draws in 1.3ms per draw. That's painfully slow. Why is it so slow? The draw_sprite assembly routine goes through about 1,800 assembly instructions to draw a single sprite! We cannot MEMCOPY if we want pixel transparency in the framebuffer. And, even if we could, it would at best double our throughput on the fast path (for x & 1 == 0).

The conclusion is, it works, but 1.3ms per draw is essentially unusable. Even a ColecoVision or a NES had more sprite power than the SD-8516 – *but not without a PPU!* The PPU provides hardware accelerated sprites, and that's just what we need here.

Wiring up the PPU

The first test was to create the sprite data manually. This is the exact same code we used for the first test above, now available as AH=$30, INT $18:

```
    ; Load sprite/tile data
    LDELM @tile_player
    LDFLD $A000           ; load it into some free area
    LDAH $30              ; This is just the import_sprite_4bpp from below.
    INT $18
    RET

tile_player:
    .bytes #16, #16
    .bytes "                "
    .bytes "      666       "
    .bytes "     66666      "
    .bytes "      666    7  "
    .bytes "       6     7  "
    .bytes "   FF88Fbb6  7  "
    .bytes "   F8FFFbb 6 7  "
    .bytes "   FF8FFb   777 "
    .bytes "   FFF8Fb    7  "
    .bytes "    F8F4        "
    .bytes "     F444       "
    .bytes "     44 44      "
    .bytes "    44   44     "
    .bytes "    66   66     "
    .bytes "   666   666    "
    .bytes "                "
```

Next, we used the PPU to load the sprite:

```
    ; Step 2: Load the converted data into PPU tile slot 0
    LDB  #0                 ; slot 0
    LDELM $A000             ; sprite data
    LDX  [ELM, +]           ; load tile width
    LDY  [ELM, +]           ; load tile height
    LDCL #0                 ; set transparent color = 0
    LDCH #4                 ; store data as 4bpp
    LDA  $0110              ; CMD_LOAD_TILE
    INT  $03
    RET
```

The calling convention will be discussed in documentation later, but you can glean it from the comments above; ELM is a pointer to the data. The header, already read into X and Y. If you didn't do this you can point ELM at the data and do:

```
  LDX [ELM, +]
  LDY [ELM, +]
```

At any rate, after wiring up the PPU to the routine, we tried this test:

```
    LDY #0
draw_yloop:
    LDX #0
draw_loop:
    ; Step 3: Draw the tile from slot 0 at (50, 50)
    LDB  #0                 ; slot 0
    LDA  $0112              ; CMD_DRAW_TILE
    INT  $03

    INC X
    CMP X, #304
    JNC @draw_loop

    INC Y
    CMP Y, #185
    JNZ @draw_yloop
```

This test attempts to draw a sprite to every valid screen location; from 0,0 to 303,183 – a total of 55,936 times. Can you guess how fast it was? The routine completed in 0.45. That's right, calibre 0.45 – 0.45 microseconds that is. Yes, that's right, it's so fast I had to repeat the test so many times because one row (~303 blits) was completing in less than a milisecond. This is very fast, and very good! At this speed, we could draw 3600 sprites per frame. But this doesn't take into account clearing the screen and syncing the frame. Let's try a test; Can we draw 32 tiles per frame, at 60fps?

Syncing the Frames

It was precisely at this time that I discovered a bug in the video code. You see, the batch executes at 241 times per second, and the video renderer executed at 60 frames per second. But due to some jitterbugs, there was an average delay introduced before the value was able to be read. After some bugfixing I was able to remove this delay. But I had to do it using super secret magic bugspray. This one was hard. I fixed it but I do not know entirely why the fix works; I only know “thats the place that needed fixing” and after quite some poking it seems to run correctly. That's the story of getting VSTEP to work correctly in assembly.

It has something to do with refresh rate. I had to just let it use the monitor's refresh rate. So if you're on 120hz for example, you get 120hz in the browser and in the SD-8516. it also means you need to know the refresh rate if you are timing your game by frames.

Blit test

Once I could sync to frame, I tried to draw one sprite per frame for five seconds. It worked. After some testing I found we could draw over 2,000 sprites *per frame* at 60fps. Too much more than that and it starts to skip frames. Still, I couldn't have expected better results:

- Up to 2048 sprites *per frame* at 60fps
- Up to 4096 sprites *per frame* at 45fps
- Up to 5600 sprites *per frame* at 30fps

Now, if you reserve 50% of your game for logic, that's 1024 sprites per frame at 60fps. The SNES could do 128 (but only 32 per scanline). This result places us firmly in the super-high end early 90s arcade board territory; Sega Y board (1988) was the first board to crack the 2000 barrier, while even later boards like the SNK Neo Geo (1990) were hardware limited to 381 sprites. It was boards like this that enabled the superscalar arcade games of the 90s. Having a microcomputer with this kind of graphics powerhouse would have been the dream of every coder.

And don't knock 30fps. There is also some value to 30fps. Games like Teenage Mutant Ninja Turtles (NES, 1989), Ghosts 'n Goblins (NES, 1986) and Contra Force (NES, 1992) ran internally at 30fps. Even SNES games like Return of Double Dragon, or N64/PS1 games (ex. Super Mario 64) would run at 30fps. These were great, legendary games; Sometimes 30fps is ok. 45fps is definitely OK.

And, for what it's worth, both Star Fox (SNES, 1993) and the Intellivision (1980!) ran at 20fps internally.

No, I couldn't be happier with this result! 2000 sprites per frame! This is great!