SD-8516 PPU

This is a short reference to the graphics capabilities of the XY-85 arcade board, an expansion board sold together with the SD-8516.

The PPU is a proprietary 32-bit central processing unit (CPU) developed for the XY-85 arcade board. It was designed to deliver high-performance framebuffer access and blit hundreds of sprites per frame, as well as provide high quality 16-bit sample playback at up to 48khz.

Introduction

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

Primitives

PPIXEL (plot_pixel)

Method	Pixels per second (WASM)
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)	660,000
PPIXEL (unrolled 1,000 times)	1,300,000

The numbers above only show that the plot rate is bound by the CPU. At one instruction per cycle, 1.28 MIPS gets ups 1.28 PIPS (pixels per second). However, once you start adding in commands that load data, process color and so on, the number can drop. A more realistic number might be closer to 300,000 pixels per second at 1 MIPS. Good enough for a bitmapped font, good enough for a couple of sprites, good enough for the early console era (Intellivision, Atari 2600).

How useful is PPIXEL if we want to push the limits? It would not be your go-to choice for anything past the late 70s era. Asteroids (1979) arcade machine is a great example. It ran on a 6502 at 1.5mhz but required a special 'digital vector generator' chip to do the heavy lifting. The 6502 simply couldn't keep up with the refresh rate even with a simple game like asteroids.

Notably, the Apple][and VIC-20 had no PPU and relied on the 6502 for all graphics. Therefore, to approximate that era, you can use PPIXEL, or just write directly to the framebuffer. Remember, if you're targeting an era that didn't use a PPU, try to avoid using the PPU to maintain the correct look and feel.

PLINE

The case for PLINE is the case for the Atari Digital Vector Generator chip used in Asteroids (1979) and

many other games.

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

Method	Lines per Second
BASIC LINE command	1,000
INT 18h draw line	2,500
BASIC using PPU via INT 0x18	15,000
BASIC using PPU direct call	16,000
PPU via INT 0x18	150,000
PPU via INT 0x03 (direct)	500,000
PLINE (PPU via opcode)	700,000

As it turns out the line drawing algorithm is also 1 cycle bound. Meaning, in the tightest loop possible, at 1.28 MIPS, 700,000 lines is our limit. This is a phenomenal number for the era, a much stronger result than a DVG chip of the era. However, if you move down to 0.3 MIPS, the numbers start making more sense.

An example would be writing an ELITE clone. Clocking down to 0.35MIPS and using the INT 0x03 interface, you would still be comfortably north of 30,000 lines per second for a game that requires at most 5,000 lines for a busy scene. This is the power of the PPU; it unlocks worlds. You could make a 60fps flicker-free clone, or you could slow it down to 20fps for extra juicy retro appeal. Why not? Ocarina of time did it. It's your choice.

PRECT and PFRECT

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

I will not comment much on this or on the next commands (PCIRCLE) except to say, they are much less frequently used, but, as far as draw primitives go they are lightning fast.

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

Again, not much to say, there is a use for this in some games, but I expect this (and RECT) will mainly be used for UI work.

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. Early consoles all had sprites, in contrast to microcomputers which didn't. The C64 changed that with 8 hardware sprites in 1982, and went on to dominate the era. After that, sprite count became a defining factor in console hardware.

- Atari 2600 5 sprites (everything is a scanline)
- Intellivision 8 sprites (no scanline limits)
- Colecovsion 32 sprites (4 per scanline)
- NES 64 sprites (8 per scanline)
- SNES 128 sprites (32 per scanline)

Our INT 18h sprite functions are mode-aware (4bpp and 8bpp) but slow. INT 18h DRAW 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
```

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 microseconds. 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

Note: The C version trebles these numbers, reaching over 7,000 sprites per frame without trying to optimize the benchmark loop.

Now, if you reserve 50% of your game for logic, that's a minimum of 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 microcomputer afficionado in the 80s/90s.

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. Other famous 30fps games include Ocarina of Time (N64, 1998), Soul Reaver (PS1, 1999), Resident Evil, Tomb Raider and GoldenEye 007. 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!

> To put this in context, the Neo Geo's 381 sprites/frame was a defining feature of its arcade hardware in 1990; games like Metal Slug and King of Fighters built their visual identity around it. Bust-A-Move. Magical Drop 3. Buttery smooth quick animation, faster than the eye could see.

The kinds of games that ran on systems betwee SNES and SNK Neo Geo arcade quality boards are all famous well known titles; OutRun, Shinobi and Altered Beast ran on Sega System 16; Capcom CPS-1 ran Street Fighter II, Final Fight, and Ghosts 'n Goblins; CPS-2 headlined Street Fighter Alpha and Marvel vs. Capcom. All of these boards could pull 200-800 sprites per frame. Namco System 2 and Taito F3 also, were both able to handle hundreds of sprites, and headlined games like Tekken, Assault (1988), Splatterhouse, and Darius Gaiden, Bubble Symphony, and RayForce.

Conclusion

A decent tile and sprite engine is the foundation of an 80s/90s console PPU and of a high end arcade board. Extensive testing must be done before I can come back and add anything to this. But, there are a few little things that may prove useful, i.e. flip transforms. As it stands, the PPU is considered ready for testing in the main system.

NES	1983	64
Sega System 16 (Outrun)	1986	~128
Capcom CPS-1 (Street Fighter II)	1988	256

Neo Geo MVS	1990	381
Sega Y Board (Galaxy Force II)	1988	~2,000
Sega System 32	1990	~1,000
3DO	1993	~1,500–2,000 (no formal limit)
Sega Saturn (2D)	1994	~5,000–8,000
PSX (2D mode)	1994	~4,000
VC-4	2026	50,000+