CSEE 4840 Embedded System Design
NES Emulator

Jeff Jaquith, Minghao Li, Zach Schuermann

Table of Contents

System Overview

The Nintendo Entertainment System (NES) can be divided into two main components, the CPU and the PPU. More specifically, the CPU has a 6502 core, an onboard audio generation and a controller interface. The second main component is the PPU which generates sprite and background tiles to produce game images from the cartridge ROM. The color palette has 64 colors and the system output is 256 by 240 pixels. In this project, the audio processing unit and sprite rendering are not implemented due to time constraint, and a single memory module is used to encompass all memory components of the cartridge. A top level file *ultranes.sv* is used to instantiate all other modules including the CPU, PPU, VRAM, clock and VGA.

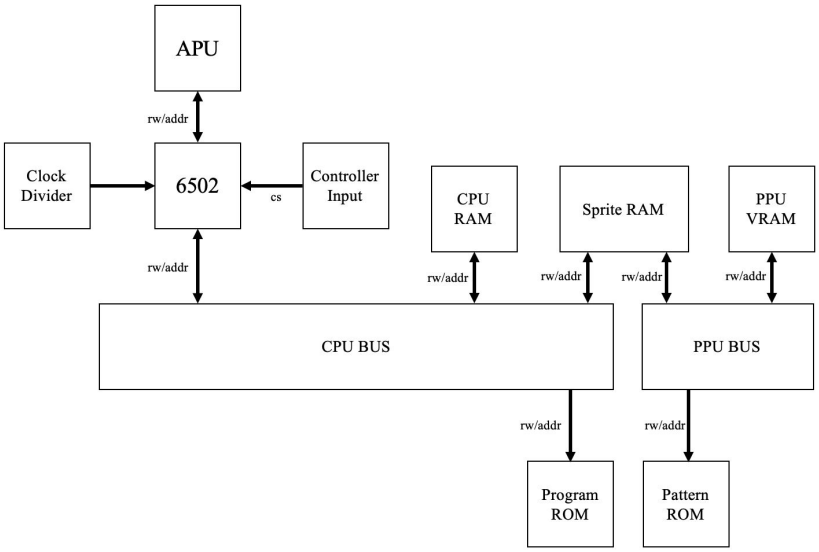


Figure 1. System Block Diagram

Memory

Name	Use	Access	Location
Character ROM 8kb	Pattern for game graphics	PPU	PPU address space
Program ROM 32kb	Game data	CPU	CPU address space
VRAM 2kb	2 nametables, 2 attribute tables	PPU	PPU address space
Sprite RAM 256 bytes	64 sprites for a frame	PPU/CPU	PPU

Palette RAM 32 bytes	sprites and background color info	PPU	PPU
System RAM 2kb	Temporary game data	CPU	CPU address space

Table 1. Memory Summary

Above is the memory access and usage table for the actual NES implementation. However, in our project we will only be using a dual port RAM with a single clock module, *vram.sv*, to encompass the entire memory space for VRAM and the ROM that is usually loaded from the game cartridge.

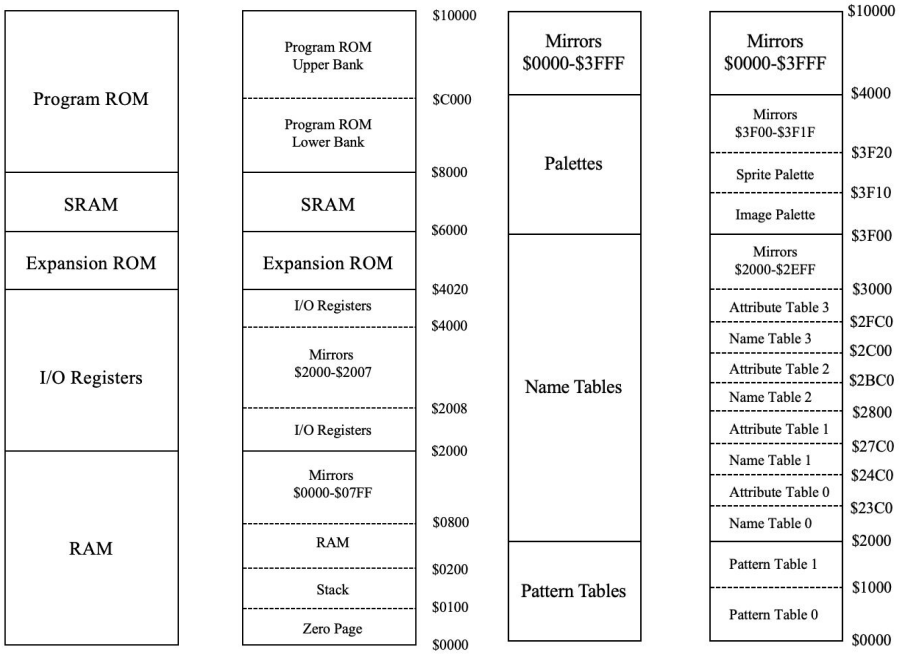


Figure 2. CPU (left) and PPU (right) Memory Map

Central Processing Unit (CPU)

On a high level, the CPU reads from the program ROM, which contains essential game information, and constructs the basic logics of the game. It then informs the PPU of how to specifically produce a pixel by pixel output on the screen. For this project, an existing 6502 core implemented in SystemVerilog is imported and connected to other components in the system.

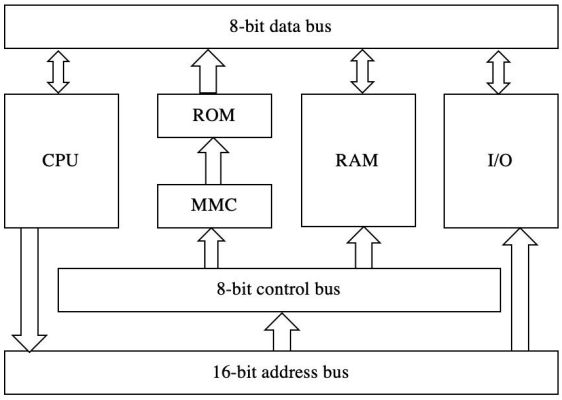


Figure 3. CPU Memory Communications

The CPU is integrated on an ASIC (labelled RP2A03), that also integrates the Audio Processor, a DMA Unit, a clock divider and a few additional pins related to controller input. The 16-bit address bus is used for the address of a requested location. It is directly generated by the CPU. The CPU is all together able to address up to 64KB of memory or memory-mapped peripherals.

The 8-bit control bus informs the connected components of whether the request is read or write. The 8-bit bidirectional data bus is used to read or write a byte to the selected address. Due to its bidirectionality, each peripheral is able to write to it at different times as specified by the address bus. To access the read-only ROM, a MMC is used for bank switching. The I/O registers are used to communicate with other components of the system.

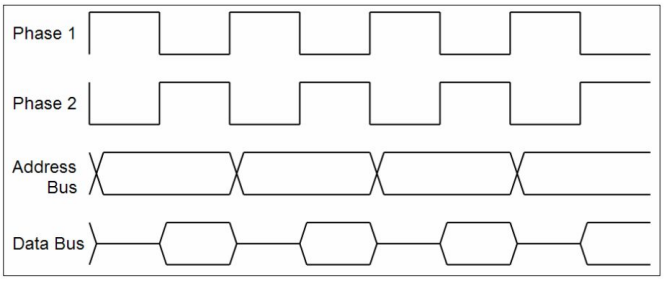


Figure 4. 6502 Bus Timing Diagram

Picture Processing Unit (PPU)

There are two modules to render the background and sprite respectively on a per pixel basis. There are 8 PPU registers and they are accessed by the CPU's address and data lines. One of the most crucial components of the PPU is its integration with the VGA. To synchronize the PPU and the VGA together, a vga module is defined for which it takes the PPU data as input and converts it to color output based on a look-up table, which takes in PPU data and outputs its VGA RGB values for respective pixels. A scanline buffer is built in VGA to hold incoming PPU data. Below is a comprehensive block diagram of the PPU and its related components.

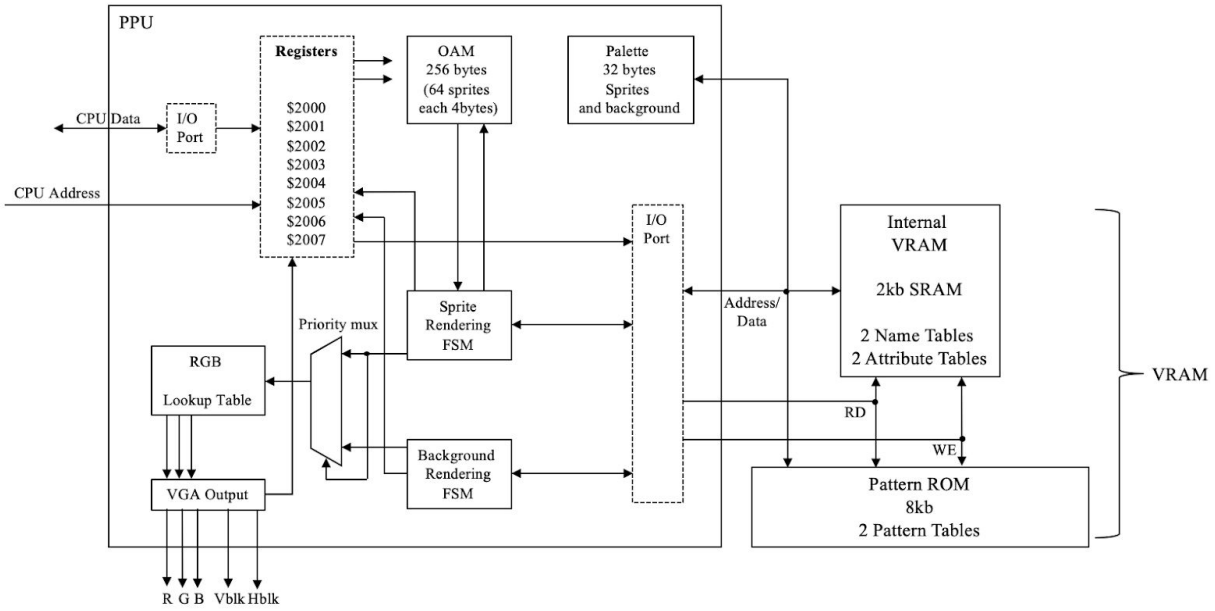


Figure 5. PPU Block Diagram

PPU Registers

Register	Details
$2000 - PPUCTRL	PPU Control Register - write Contains flags that describe PPU operation **LSB** **NN -** Designates name table base address (0=2000, 1=2400, 2=2800, 3=2C00) **I -** VRAM address increment per CPU read/write of PPUDATA ($2007) (0: increment 1, going across, 1:increment 32, going down) **S -** Sprite table base address (0=0000, 1=10000) *only in 8x8 mode **B -** Background pattern table address (0=0000, 1=1000) **H -** Sprite size (0=8x8, 1=8x16) **V -** PPU as master or slave (0: read from EXT pins, 1:output color to EXT pins) **N -** 1: sets a non maskable interrupt at start of BLANK **MSB** 2 LSBs are MSBs of scrolling location (bit 0 - adds 256 to X scroll, bit 1 - adds 240 to Y scroll) After power on writes to register are ignored for 30k cycles
$2001 - PPUMASK	PPU Mask Register - write Controls the rendering of sprites and background tiles **LSB** **G -** Greyscale (0: normal, 1: grey)

	m - 1: Show background is leftmost 8 pixels, 0:hide **M -** 1: Show sprite in leftmost 8 pixels, 0: hide **b -** 1: Show Background **s -** 1: Show Sprite **R -** 1: Emphasize red **G -** 1: Emphasize green **B -** 1: Emphasize blue **MSB** Bits 1&2 (m&M) enable rendering for leftmost 8 pixel columns - this is useful for scrolling (when you want parital sprites or tiles to scroll in from the left) Bits 3&4 (b&s) Render background or sprite respectively If changes to VRAM outside of VBLANK set b&s to 0.
$2002 - PPUSTATUS	PPU Status Register - read **LSB** **(0-4)** LSBs previously written into PPU Reg **O -** Sprite overflow, flag is set during sprite evaluation, cleared at second tick of pre-render line **S -** Set when non zero pixel of sprite zero overlaps non zero background pixel. Cleared at second tick of pre render line **V -** Set at tick 1 of of line 241 (1 for in VBLANK, 0 for not). Cleared after reading this register and second tick of pre-render line. Reading does not clear O&S
$2003 - OAMADDR	OAM Address - write Write address of OAM to access. Set to 0 during 257-320 ticks of pre render and visible scan lines (Sprite loading interval) Value at tick 65 of visible scan lines determines where in OAM sprite evaluation starts regardless of byte type and every byte following is interpreted accordingly. This effectively hides the sprites before the first address is accessed.
$2004 - OAMDATA	OAM Data - read/ write Writes will increment OAMADDR after the write. Reads do increment, occur during VBLANK or forced blanking.
$2005 - PPUSCROLL	PPU Scrolling position register - write x2 Typically written to during VBLANK (can be modified during rendering sp split the screen) Tell PPU which pixel in the nametable being used is the top left pixel. Nametable is selected using NN in PPUCTRL.
$2006 - PPUADDR	PPU Address register - writex2 Specifies the 16-bit address in VRAM that $2007 will use CPU writes to VRAM through PPUADDR and PPUDATA

	Upper byte is written first. BYTE 1: upper 8-bit of effective address BYTE 2: lower 8-bit of effective address
$2007 - PPUDATA	PPU Data - read / write After access I bit in PPUCTRL determines how much to increment address. Only accessed during VBLANK or forced blanking. After access needs to reload scroll position.
$4014 - OAMDMA	OAMDMA register - write Located on the CPU. Used to upload 256 bytes from CPU $XX00-4XXFF to PPU OAM. Transfer takes 513/514 cycles during which the CPU is suspended. Should take place during VBLANK, writes through OAMDATA or not suited.

Table 2. PPU Registers

Background Rendering

Name tables contain 8x8 pixel tiles for displaying graphics, it is the layout of a frame's background. In total, the name tables contain 32x30 tiles (256x240 pixels). Each tile contains a single byte in PPU memory. It only holds the tile number of the data that is kept in the pattern table

The pattern table holds the actual 8x8 tile data and also the lower 2 bits of the 4 bit color matrix needed to access all 16 colors. It has the static info from the ROM that PPU can read only.

```
 VRAM    Contents of                    Colour
 Addr   Pattern Table                   Result
------ ---------------                 --------
$0000: %00010000 = $10 --+             ...1.... Periods are used to
  ..   %00000000 = $00   |             ..2.2... represent colour 0.
  ..   %01000100 = $44   |             .3...3.. Numbers represent
  ..   %00000000 = $00   +-- Bit 0     2.....2. the actual palette
  ..   %11111110 = $FE   |             1111111. colour #.
  ..   %00000000 = $00   |             2.....2.
  ..   %10000010 = $82   |             3.....3.
$0007: %00000000 = $00 --+             ........

$0008: %00000000 = $00 --+
  ..   %00101000 = $28   |
  ..   %01000100 = $44   |
  ..   %10000010 = $82   +-- Bit 1
  ..   %00000000 = $00   |
  ..   %10000010 = $82   |
  ..   %10000010 = $82   |
$000F: %00000000 = $00 --+
```

Figure 6. Pattern Table

In the attribute table, each byte represents a 4x4 group of tiles on the screen

```
+------------+------------+
|  Square 0  |  Square 1  |  #0-F represents an 8x8 tile
|   #0  #1   |   #4  #5   |
|   #2  #3   |   #6  #7   |  Square [x] represents four (4) 8x8 tiles
+------------+------------+   (i.e. a 16x16 pixel grid)
|  Square 2  |  Square 3  |
|   #8  #9   |   #C  #D   |
|   #A  #B   |   #E  #F   |
+------------+------------+

 Attribute Byte
   (Square #)
-----------------
    33221100
    ||||||+--- Upper two (2) colour bits for Square 0 (Tiles #0,1,2,3)
    ||||+----- Upper two (2) colour bits for Square 1 (Tiles #4,5,6,7)
    ||+------- Upper two (2) colour bits for Square 2 (Tiles #8,9,A,B)
    +--------- Upper two (2) colour bits for Square 3 (Tiles #C,D,E,F)
```

Figure 7. Attribute Table

In the palette table, every frame has its own subset of palette of the system palette. And the frame palette is dynamic meaning that two frames could have different sets of palettes that they use to produce colors. CPU sends palette entries to the PPU. In a frame palette, there are 8 palette groups each with 4 colors. 0-3 are for background and 3-7 are for sprites.

The color information at a specific pixel is determined by the priority mux shown below.

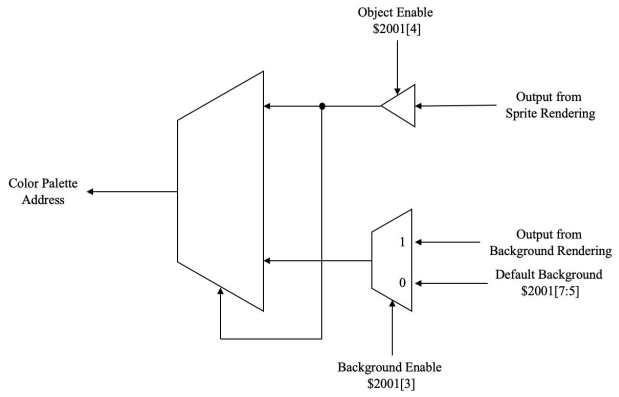


Figure 8. Priority Mux

Sprite Rendering

There are 64 sprites displayed in any given frame . The High-level memory is constructed by the CPU during vertical blank and there are only 8 sprites per scanline due to time constraint. Sprite buffers are used for pattern data for the sprite, color attributes, priority information, and an exact horizontal coordinate.

In summary, sprite rendering follows the below steps:

1. PPU scans through the y coordinates in sprite RAM to determine whether the sprite should be displayed on the next scan line
2. If so store this data to the secondary OAM or store the sprite's index number. The secondary OAM is only used to store sprites for the next scanline.
3. Fetch and store pattern info in sprite buffers (shift registers) to make sure no conflict occurs with range checking
4. Pixels output uses the sprite buffers which contain two shift registers and a horizontal coordinate counter. The background gets drawn to a separate buffer.
5. The coordinate counter tells whether the sprite should display and and clocks the shift registers to output data.

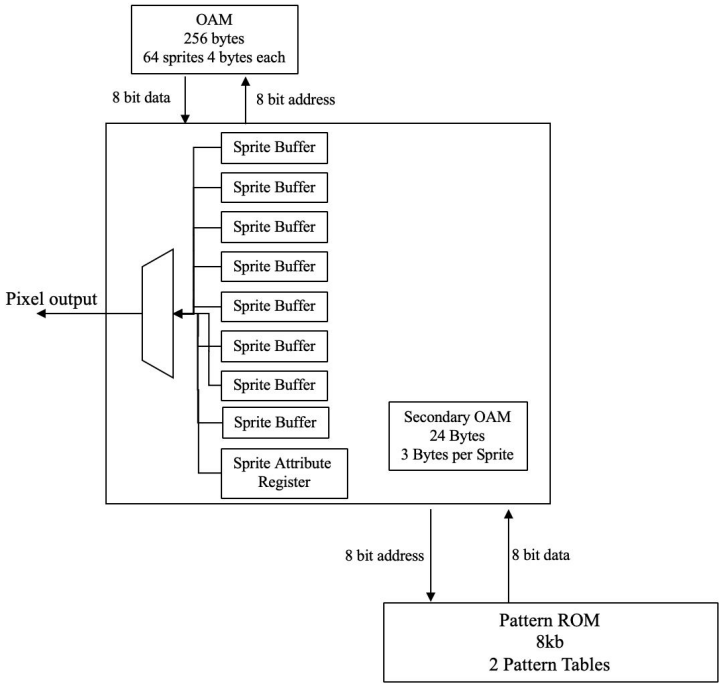


Figure 9. Sprite Renderer

PPU Rendering Example - Donkey Kong

Nametable:

- Tiles of Donkey Kong: nametable is the layout of the frame's background
  - (00,00) upper left to (1F,1D) bottom right
- Each tile contains a single byte in PPU memory
- The numbers 24 and 62 shown below are just indices into the pattern table

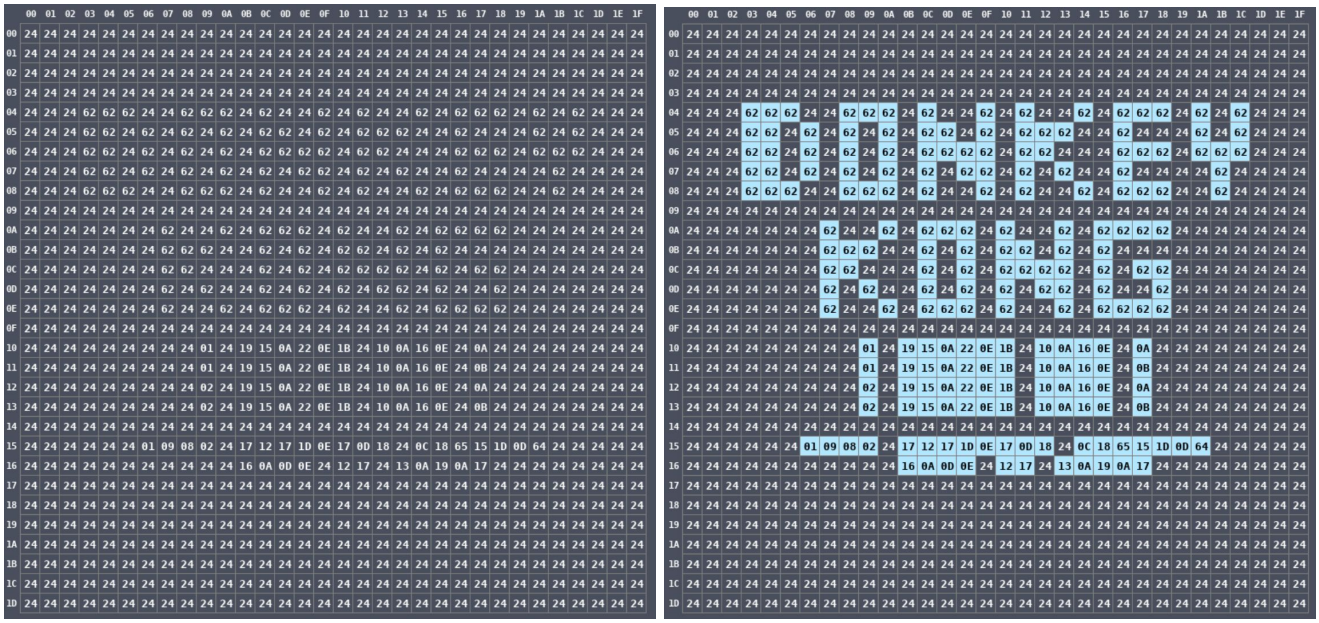
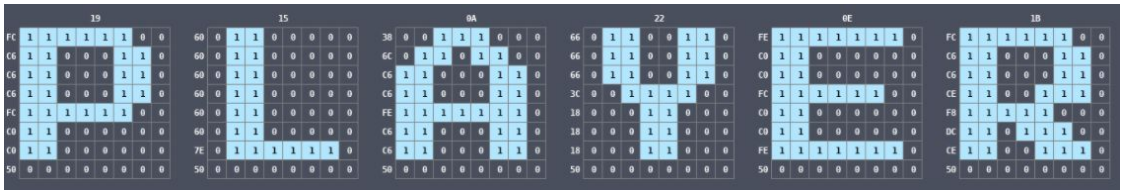


Figure 10. Tiles of Donkey Kong

Pattern Table:

- For tile at position (09,10) it has index 01
  - In the pattern table at 01 we have 16 bytes and separating them into low and high bytes to get
  - 0:7 18 38 18 18 18 18 7E 00
  - 8:F 00 00 00 00 00 00 00

Figure 11. Combining Low and High Bitmap of Pattern Table



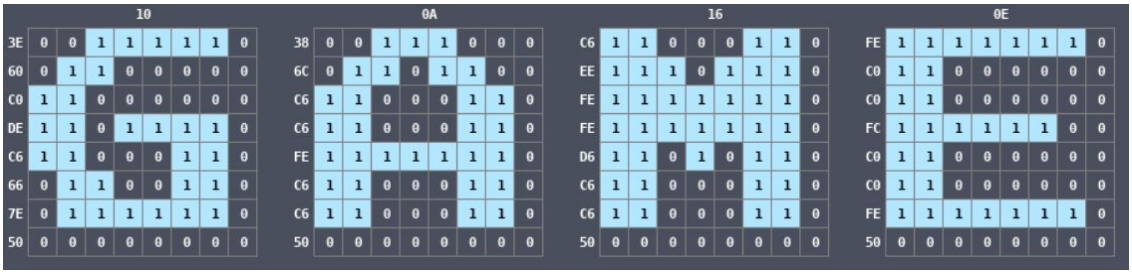


Figure 12. Example of Pattern Table Combination

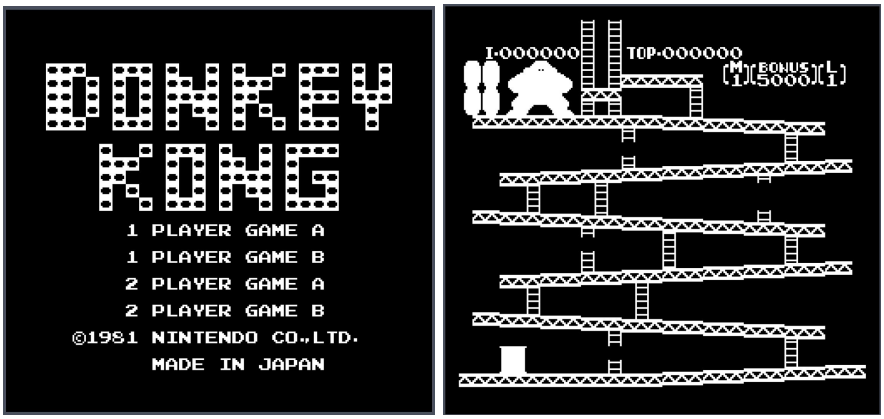


Figure 13. Donkey Kong Example Pattern Table Combination Without Palette

System Palette:

- Define by NES a finite set of colors numbered from 00 to 3F

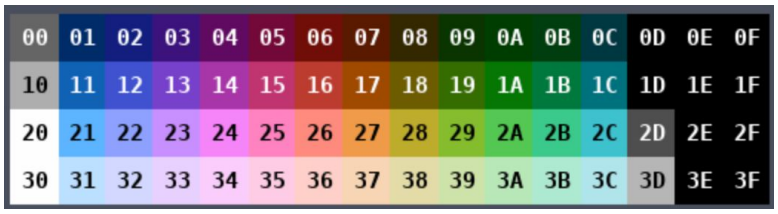


Figure 14. System Color Selection

Frame Palette:

- Every frame has its own subset of palette of the system palette
- Frame palette is dynamic
  - o CPU sends palette entries to the PPU so different frame can use different frame palette
- 8 palettes and each sub palette group has four colors
  - o 0-3 are for backgrounds and 4-7 are for sprites

Figure 15. Palette RAM

- For the oil drum

- (04, 19) (05,19) (04,1A) (05,1A)

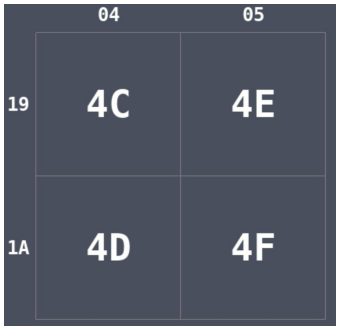


Figure 16. Nametable For Oil Drum

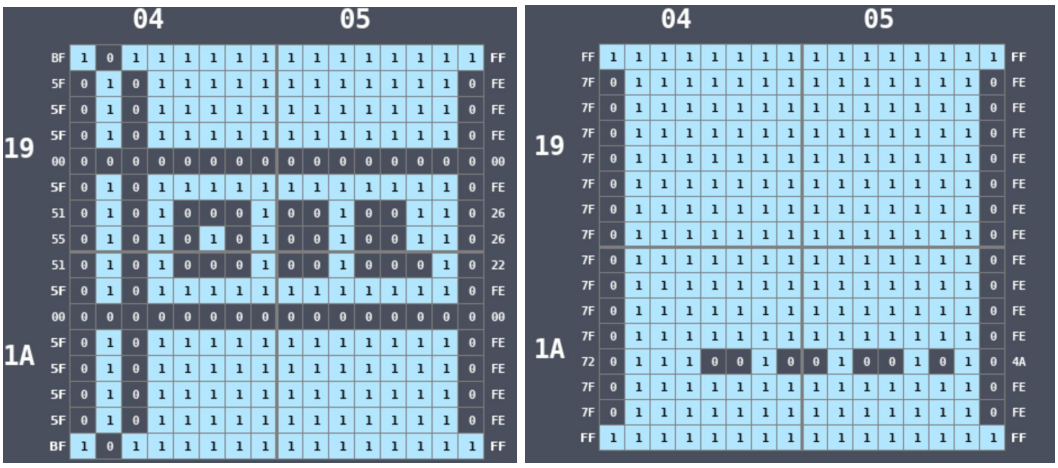


Figure 17. Pattern Table Low and High Bitmap for Oil Drum

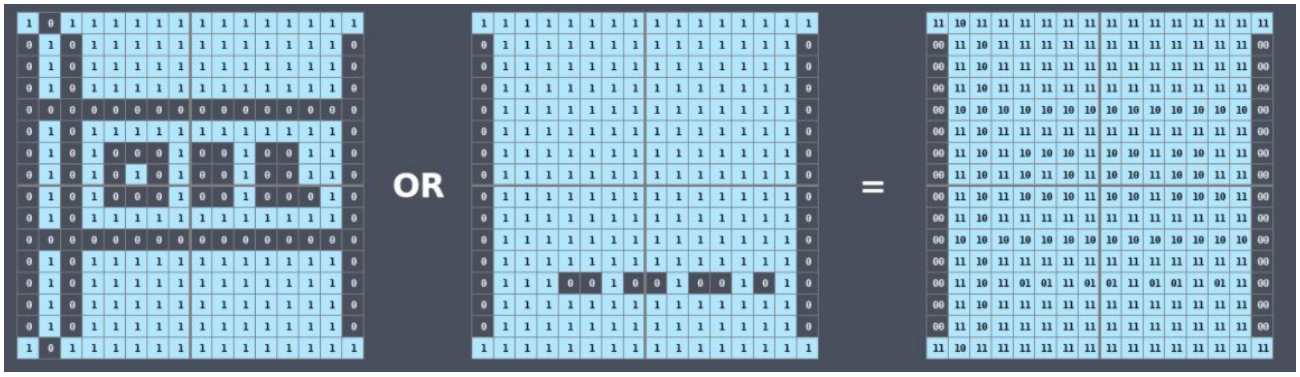


Figure 18. Combining Low and High Bitmap

- There are values between 00 and 11
- This is the index into the frame palette

Figure 19. Palette RAM

- We will use the attribute table to know which palette we are accessing
- In this case we are accessing palette table 0

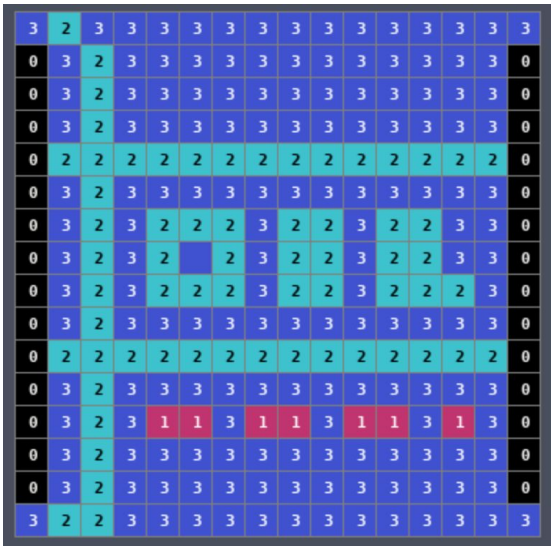


Figure 20. Coloring After Accessing the Palette Table

Attribute Table:

- The tiles are divided into blocks, each block is a 4x4 tiles
- Continuing with the oil drum example we have
  - In block (1,6)

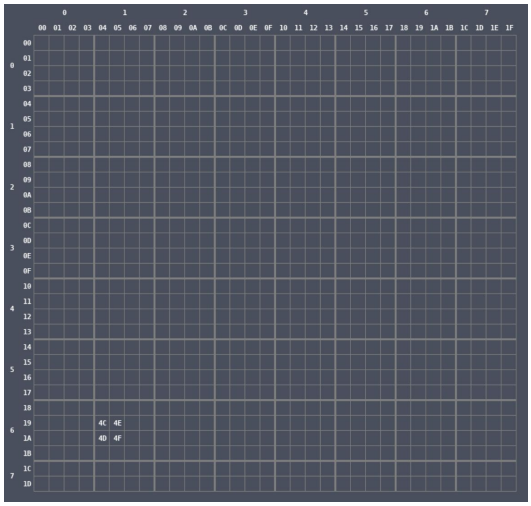


Figure 21. Divided Block

- A block above is divided into 4 tiles, 0 1 2 3
- In the attribute table each block is a single byte
  - Four 2 bit value
  - Quad 0 is bit 0 and 1
  - Quad 1 is bit 2 3
  - Quad 2 is bit 4 5
  - Quad 3 is bit 6 7
- In our example, at the location of the oil drum the byte is 0 so

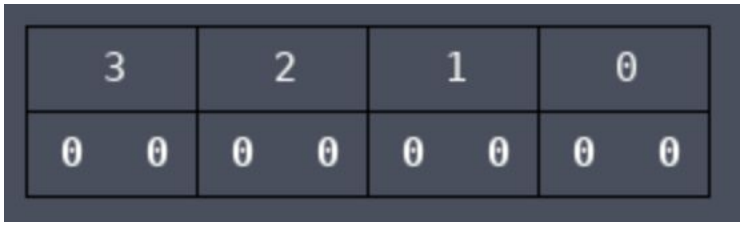


Figure 22. Index into the Palette RAM

- Here is an example of Donkey Kong

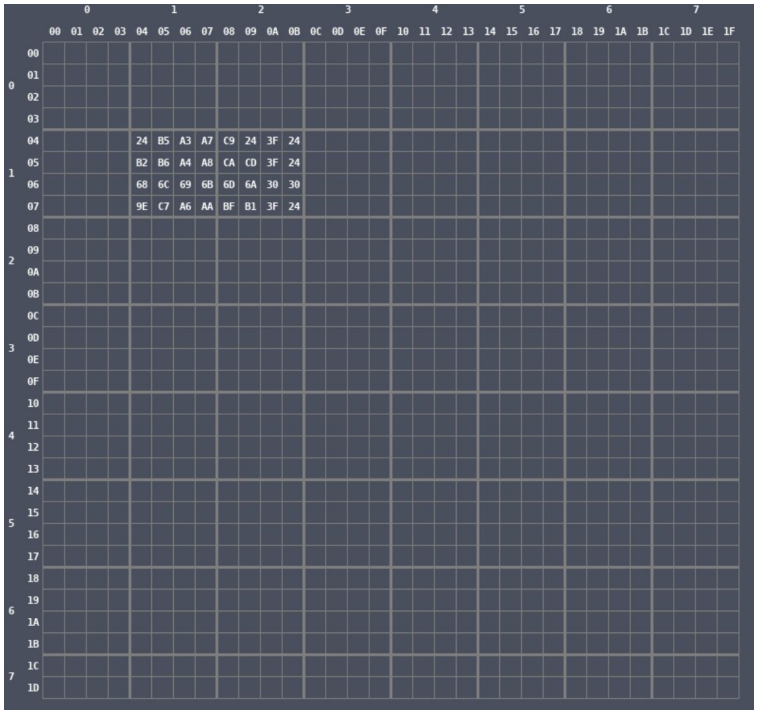


Figure 23. Example of Donkey Kong in Nametable

- Looking up values in the pattern table we have the following

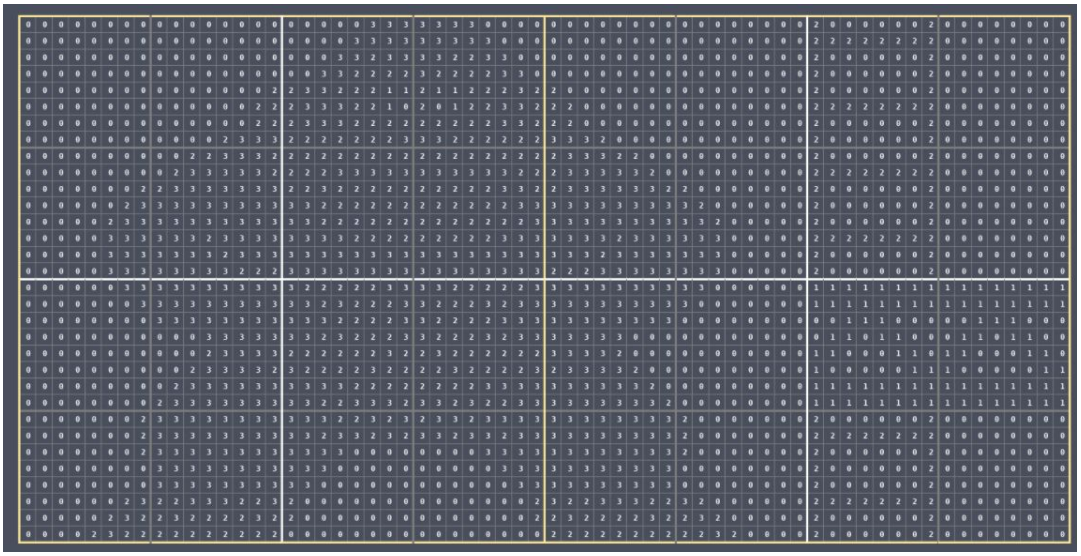


Figure 24. Donkey Kong Divided into Blocks

- For block (1,1) in the attribute table we read AA
  - Each region of the four regions will use palette 2

3		2		1		0	
1	0	1	0	1	0	1	0

Figure 25. Block (1,1) in Attribute Table

- For block (2,1) in the attribute we read 22
  - Quad 0 and 2 will be palette 2 and 1 and 3 will be palette 0

3		2		1		0	
0	0	1	0	0	0	1	0

Figure 26. Bock (2,1) in Attribute Table

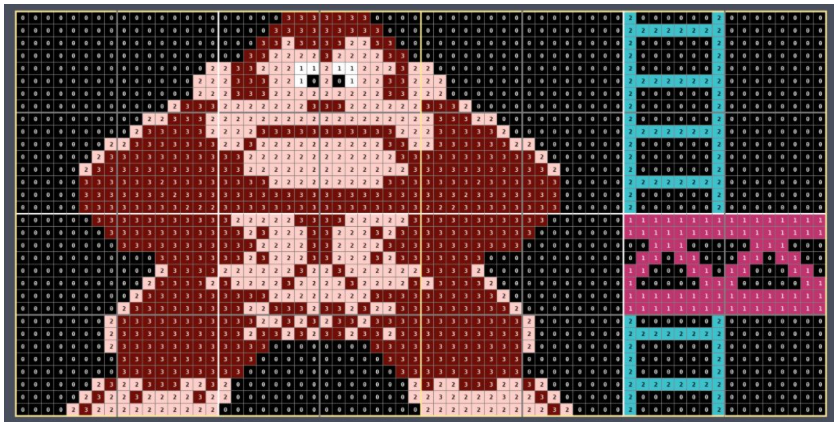


Figure 27. Donkey Kong Rendered using Palette RAM

Sprites:

- a sprite has 4 bytes that can be accessed in the OAM: y pos, x pos, tile index and attribute.
- For Mario

	Y	Tile	Att	X
0	C7	4	0	3
1	CF	5	0	30
2	C7	6	0	38
3	CF	7	0	38

Figure 28. Mario Sprite in OAM

- Using the same procedures for background for tile IDs and combining the bit maps

0	0	0	0	0	3	3	3	3	0	0	0	0	0	0	0
0	0	0	0	3	3	3	3	3	3	3	3	0	0	0	0
0	0	0	0	1	1	1	1	2	2	0	0	0	0	0	0
0	0	0	1	2	2	1	2	2	1	2	2	2	0	0	0
0	0	0	1	2	2	1	1	2	2	1	2	2	2	0	0
0	0	1	1	1	2	2	2	2	1	1	1	1	0	0	0
0	0	0	0	0	2	2	2	2	2	2	2	0	0	0	0
0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0
0	0	0	1	1	1	1	3	3	1	1	0	0	0	0	0
0	0	0	1	1	1	3	3	2	3	3	0	0	0	0	0
0	0	0	1	1	1	1	3	3	3	3	3	0	0	0	0
0	0	0	3	1	2	2	2	3	3	3	3	0	0	0	0
0	0	0	3	3	2	2	3	3	3	3	3	0	0	0	0
0	0	0	3	3	3	3	0	3	3	3	0	0	0	0	0
0	0	0	1	1	1	0	0	1	1	1	0	0	0	0	0
0	0	0	1	1	1	1	0	1	1	1	1	0	0	0	0

Figure 29. Mario in Nametable

- For Mario, the attribute byte is 00
  - o so we use palette 0 and in the palette frame it's palette 4

Figure 30. Palette RAM

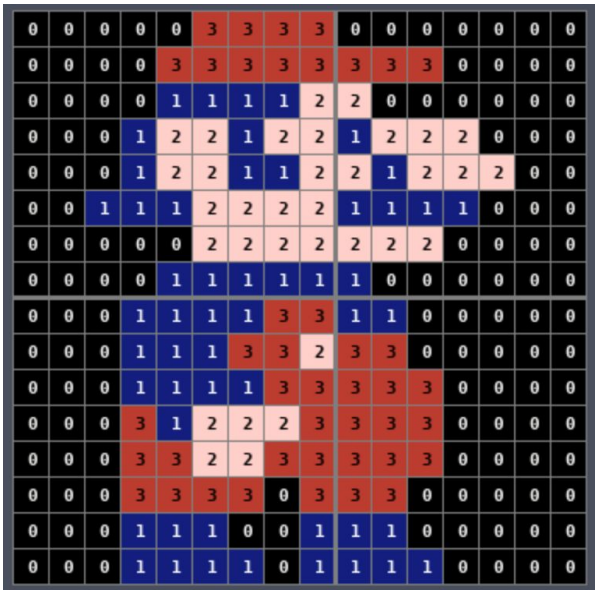


Figure 31. Mario Sprite Rendered

PPU Rendering Figures in Summary

- PPU renders 262 scan lines per frame
  - 240 visible scan lines
  - 20 fetching data (vblank)
  - 2 dummy
- Only can write one pixel per PPU cycle
  - Takes 341 PPU cycles per scanline
  - 256 for rendering; remaining are used to fetch data from nametables, etc.
  - (2 clock cycles per pfetch, PPU multiplexes bottom 8 VRAM Address pins to also use as data pins)
- For each frame:
  - -1 scanline: prefetch tile info for first two tiles. No pixel rendering for this scanline
  - 0-239 scanline: render background and sprite. The program does not access PPU memory at this time unless rendering is off.
  - 240 scanline: idle. Vblank is set after this scanline.
  - 241-260 scanline: vblank lines, CPU can access VRAM because PPU makes no memory access during these scanlines.

- For each visible scanline (0-239):
  - 0 cycle: idle
  - 1-256 cycle: visible pixels. Each memory access takes 2 PPU cycles and each tile needs 4 for nametable, attribute table, pattern table low and pattern table high.
    - Output pixel based on VRAM
    - Prefetch next tiles. PPU can only fetch an attribute byte every 8 cycles.
    - Sprite evaluation for next scanline
  - 257-320: prefetch tiles data for sprites on the next scanline
  - 321-336: prefetch the first two tiles for the next scanline and loaded to the shift registers
  - 337-340: unknown fetches

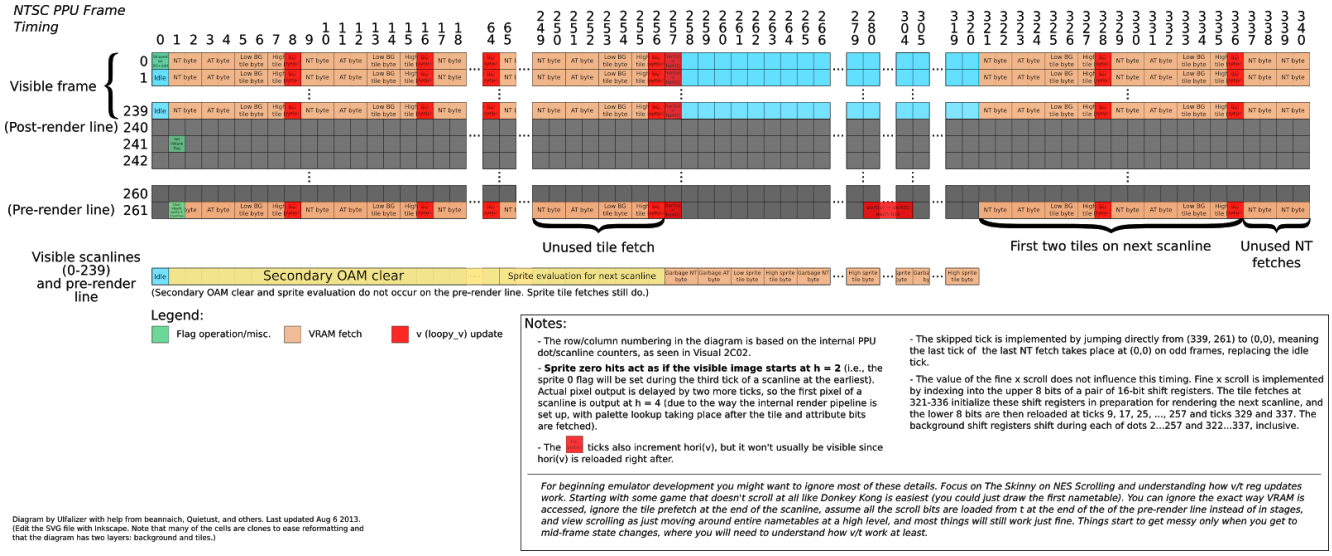


Figure 32. Timing Diagram Form NESDev Wiki

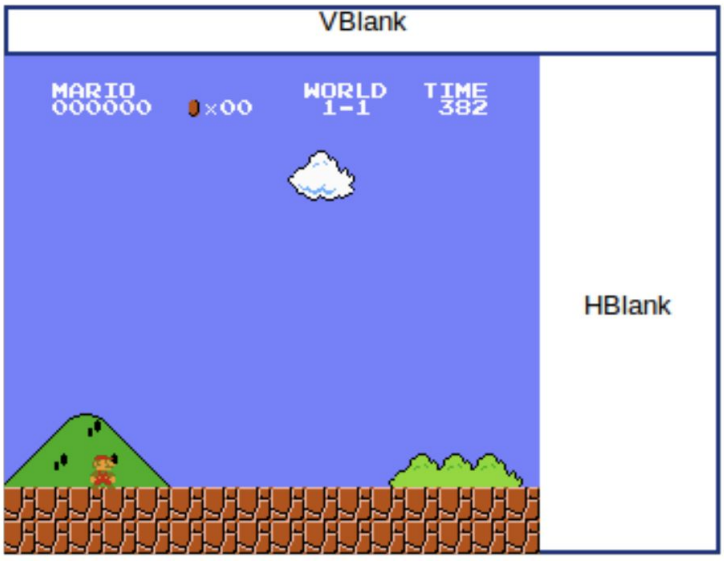


Figure 33. Blank Areas Used During CPU Cycles/Data Fetch

Linux Userspace Utilities

There are three main components:
1. Avlon bus interface to the FPGA
2. Linux device driver for memory-mapped access to avalon-bus
3. Userspace utility to issue IOCTL's to modify RAM/ROM onboard FPGA

We have an installer script to build device driver, install kernel module, and install pre-compiled userspace utility defined as 'ultranes' binary show below.

```
$ ultranes                          # print current value on address bus
$ ultranes reset 1                  # set CPU reset high
$ ultranes reset 0                  # set CPU reset low
$ ultranes load cpu bin/test.hex    # load test.hex ROM into CPU RAM
$ ultranes load ppu bin/vtest.hex   # load vtest.hex ROM into PPU RAM
$ ultranes load dk bin/dk.nes       # load dk.nes ROM into CPU+PPU RAM
$ ultranes load mario bin/mario.nes # load mario.nes ROM into CPU+PPU RAM
$ ultranes write 55 128             # write value 55 to address 128
$ ultranes help                     # call in backup
```

Figure 32. Ultranes binary

Clocking/Timing Figures

The total clocking scheme is facilitated via a global clock (50MHz) and respective clock enables. For the VGA, it is running at double the resolution of the PPU (so 4x the number of pixels), the PPU is 4 times slower than the VGA so they are in sync. Each PPU frame will take 89,342 PPU cycles and each VGA frame will take 357,368 VGA cycles (exactly 4x more). To ensure correct timing, a clock module is defined and several clock enables are fed into other modules including the CPU, PPU and VGA.

- 50 MHz global clock
- 25 MHz VGA clock (50/2)
- 6.25 MHz PPU clock (50/8)
- 2.083 MHz CPU clock (50/24)

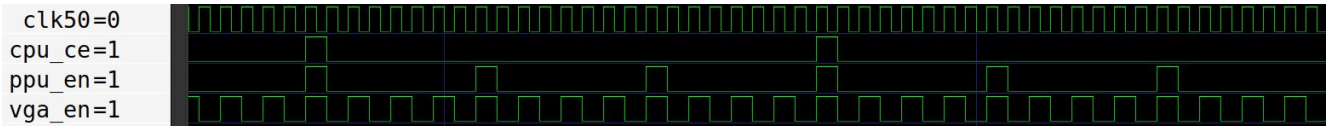


Figure 33. Timing Simulation

Arbitration of Shared Resources

CPU/PPU share memory, CPU updates mem (VRAM) during VBlank. A dual-port block RAM is used for PPU and CPU to access memory separately.

Hardware/Software Interface

The hardware/software interface exists between the FPGA implementation of the NES and the Linux host which will load ROM's (game cartridges) onto the board.

Resource Requirements

The DE1-SoC has 64MB SDRAM, far greater than any combination of NES + ROM cartridges with additional RAM (no more than 1MB). Furthermore, the original 6502 had ~3,200 transistors, while our FPGA has 85,000. The 6502 core we plan to use occupied only 8% of the flops and 7% of the LUTs in the Xilinx xc3s500e FPGA.

References

https://www.quora.com/q/oefyspdckxhlovmr/How-NES-Graphics-Work-Pattern-tables
https://austinmorlan.com/posts/nes_rendering_overview/
http://web.mit.edu/6.111/www/f2004/projects/dkm_report.pdf
http://nesdev.com/NESDoc.pdf
https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_altpll.pdf
http://www.aholme.co.uk/6502/Main.htm
http://tuxnes.sourceforge.net/nestech100.txt - NES Documentation
http://web.mit.edu/6.111/volume2/OldFiles/www/f2019/projects/dklahn_Project_Final_Report.pdf

Appendix

Sprite DMA Unit: For updating to the internal PPU memory, there are memory mapped registers at $2003 and $2004. $2003 is used to set the internal address, $2004 writes a value with auto-increment.
Since most games would want to update all 256 bytes of sprite data in each frame, the 2A03 integrates a unit that can halt the CPU and copy one memory page to $2004 directly.

```
             -------
      +5V -- |36   72| -- GND
 CIC toMB <- |35   71| <- CIC CLK
CIC toPak -> |34   70| <- CIC +RST
   PPU D3 <> |33   69| <> PPU D4
   PPU D2 <> |32   68| <> PPU D5
   PPU D1 <> |31   67| <> PPU D6
   PPU D0 <> |30   66| <> PPU D7
   PPU A0 -> |29   65| <- PPU A13
   PPU A1 -> |28   64| <- PPU A12
   PPU A2 -> |27   63| <- PPU A10
   PPU A3 -> |26   62| <- PPU A11
   PPU A4 -> |25   61| <- PPU A9
   PPU A5 -> |24   60| <- PPU A8
   PPU A6 -> |23   59| <- PPU A7
CIRAM A10 <- |22   58| <- PPU /A13
  PPU /RD -> |21   57| -> CIRAM /CE
    EXP 4    |20   56| <- PPU /WR
    EXP 3    |19   55|    EXP 5
    EXP 2    |18   54|    EXP 6
    EXP 1    |17   53|    EXP 7
    EXP 0    |16   52|    EXP 8
     /IRQ <- |15   51|    EXP 9
  CPU R/W -> |14   50| <- /ROMSEL (/A15 + /M2)
   CPU A0 -> |13   49| <> CPU D0
   CPU A1 -> |12   48| <> CPU D1
   CPU A2 -> |11   47| <> CPU D2
   CPU A3 -> |10   46| <> CPU D3
   CPU A4 -> |09   45| <> CPU D4
   CPU A5 -> |08   44| <> CPU D5
   CPU A6 -> |07   43| <> CPU D6
   CPU A7 -> |06   42| <> CPU D7
   CPU A8 -> |05   41| <- CPU A14
   CPU A9 -> |04   40| <- CPU A13
  CPU A10 -> |03   39| <- CPU A12
  CPU A11 -> |02   38| <- M2
      GND -- |01   37| <- SYSTEM CLK
             -------
```