Introduction & Summary

Nintendo Entertainment System is game console in 70s. This project explores and recreates its original hardware design.

NES composes few parts that's need to be implemented. It has a 8-bit **CPU** for game instructions. **PPU** (Pixel Processing Unit) for rendering game graphics to TV signals. (NTSC in Japan/US, PAL in Europe). **pAPU** (pseudo Audio Processing Unit) for generating 8-bit audio. Internal **MMU** for mapping the CPU and PPU memory.

The NES uses game cartridge that will be plugged into the NES. Cartridge contains game instruction ROM (**PRG-ROM**), and game character / pattern ROM (**CHR-ROM**). Some game might has some save ram (**SRAM**) for game state saving, and there are few other memory that might appear in the cartridge. These memory are mapped using cartridge's internal **Mapper** to the CPU and PPU memory, respectively.

Modern software simulator usually uses **iNES** format to encode the game cartridge memory into single file. The details of **iNES** file will be explained later.

This project recreates the CPU part and most of the PPU parts. pAPU is left undone. The game cartridge is flashed directly into ROM using Vivado coe file. More could be done in the future as listed in the conclusion.

CPU

Related Information

NES uses a variant of popular MOS 6502 at that time. It's called 6502-2A03. Unlike the standard 6502, it removes the **BCD** (binary coded decimal) to cut down the price. It also supports the 8-bit audio output inside the CPU.

MOS-6502 is an 8-bit CISC CPU. There is a good reference guide with (de)assembler in Masswerk website, the C74 Project also provides a comprehensive guide to 6502 TTL implementation.

In 6502, every instruction varies from 1 byte to 3 bytes, while the cycles each execution could take ranges from 2 cycles to 7 cycles at max. The first byte of every instruction is the opcode. 6502 only uses a bit more than half of all possible 256 opcode values, so there are some *illegal* opcodes. If an illegal opcode is encountered, there won't be an exception logic like what happens in x86 architecture. Instead, an undefined behavior could happen, depending specifically on the hardware implementation of the chip. There are some stable and somewhat used illegal opcodes, which could done some operation more quickly than normal instruction could not (a similar taste like the Mode X in VGA graphics).

6502 has far less registers than x86 or RISC architecture CPUs. It exposes users these registers (all 8-bit):

- **A** register, the Accumulator.
- **X/Y** registers, two indexing register.
- **SP** registers, stack pointer. Details about CPU memory map explained later.

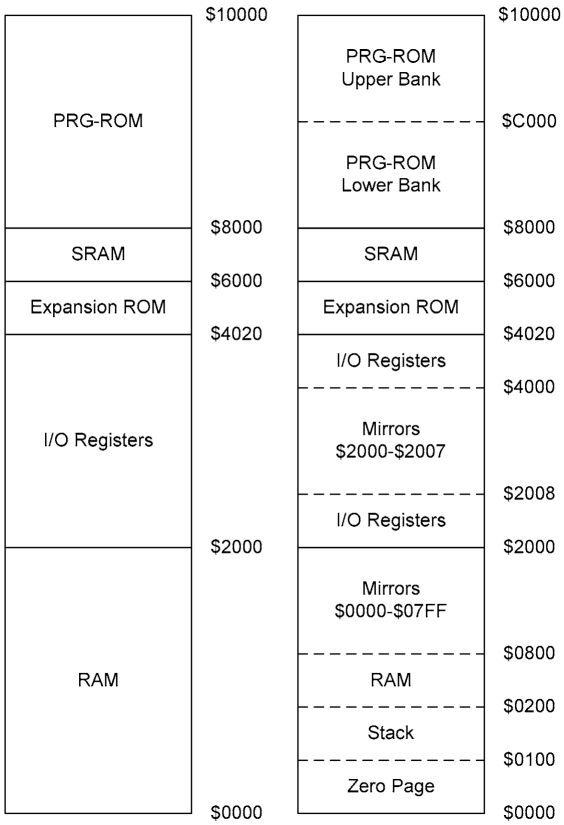
- **P** registers, the status flag register, it has 7 flags, though only 5 flags are used in 6502-2A03.
  - **N**, indicates last ALU operation is negative or not
  - **Z**, indicates last ALU operation is zero or not
  - **I**, indicates whether interrupt is disabled (1) or not (0).
  - **V**, indicates whether last arithmetic operation has overflow or not.
  - **C**, indicates whether last ALU has carry or not
  - ***D***, indicates whether BCD is enabled (1) or not. Not implemented in 6502-2A03
  - There is also 1 pseudo flag, which doesn't not have presence in hardware, but the flag will have correct value when the **P** register is stored into the memory.
  - **P**: NV1BDIZC (from MSB to LSB)

There are few internal registers used by CPU, they are:

- **PCL** (Program Counter Low), **PCH** (Program Counter High) from tracking the address of next instruction.
- **DPL** (Data Pointer Low), **DPH** (Data Pointer High) for temporarily store the address.
- **B** register: temporarily store one 8-bit value.
- **IC** Flag (Internal Carry Flag), used for branching and keeping the carry when addition to **DPL/PCL** overflows.

In 6502, 11 different addressing mode has been provided just like x86. Since it's an 8 bit CPU with 16-bit address space, few more a bit *strange* addressing mode has been provided. Each instruction will support some addressing modes out of 11 modes specified. More details is in the listed reference, or the excel sheet provided with the report.

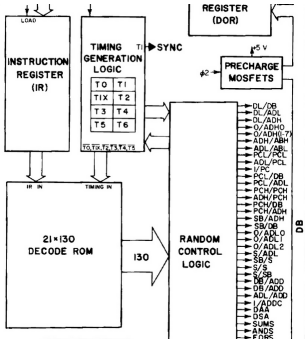
6502 has a memory map looks like below (adapted from NESdoc):



As said, it has a 16-bit, 64kB address space. The upper PRG-ROM, SRAM, Expansion ROM address range is all forwarded by MMU to the internal **Mapper** in the game cartridge. The address from `$0000` to `$4020` is mapped internally by MMU. Each section in these range are:

- Zero page: used by 6502 zero page addressing modes, which usually takes fewer cycles.
- Stack: the stack pointer **SP** as mentioned is 8-bit register, when accessing the stack memory, address of `{8'h01, SP}` is used
- RAM: the usable memory area.
- Above memory is mirrored three times in `$0800 - $2000`.
- IO registers: the IO registers between `$2000 - $2007` is PPU control registers. (It is mirrored till to `$4000`) the register between `$4000-4020` includes pAPU and test-mode control registers. Test-mode control register, according to the reverse engineering, is used for CPU debugging process, though its detail is unknown.

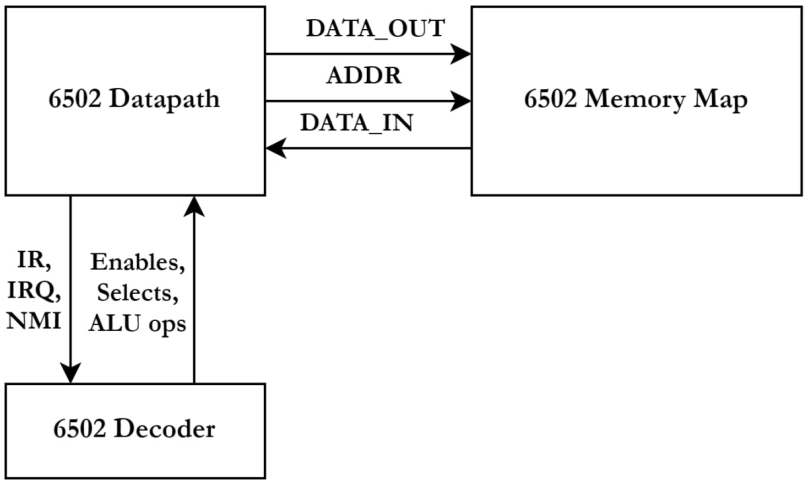
The original 6502 uses a Programmable Logic Array (PLA) to decode the instruction into some internal instruction (they are not calling this microinstruction). The PLA will compare the instruction (with a internal timing state) with some pre-existing masks. If the instruction timing state combination is accepted one mask, it will fire the corresponding internal instruction. All the internal instruction are ORed together to formal the final decoded internal instruction. Since the decoded instruction is a series of control signals (like enables / mux selects), when an illegal opcode is encountered, it will fire multiple internal instruction which normally should not be in combination, and when OR-ing these, some interesting process could go on in the CPU. This is explains why some illegal opcode could work and even be useful. More details on [How illegal opcode really works]



> The PLA decoder in 6502 design (Decode ROM and Random Control Logic here). Image from originally Apple II computer

Implementation

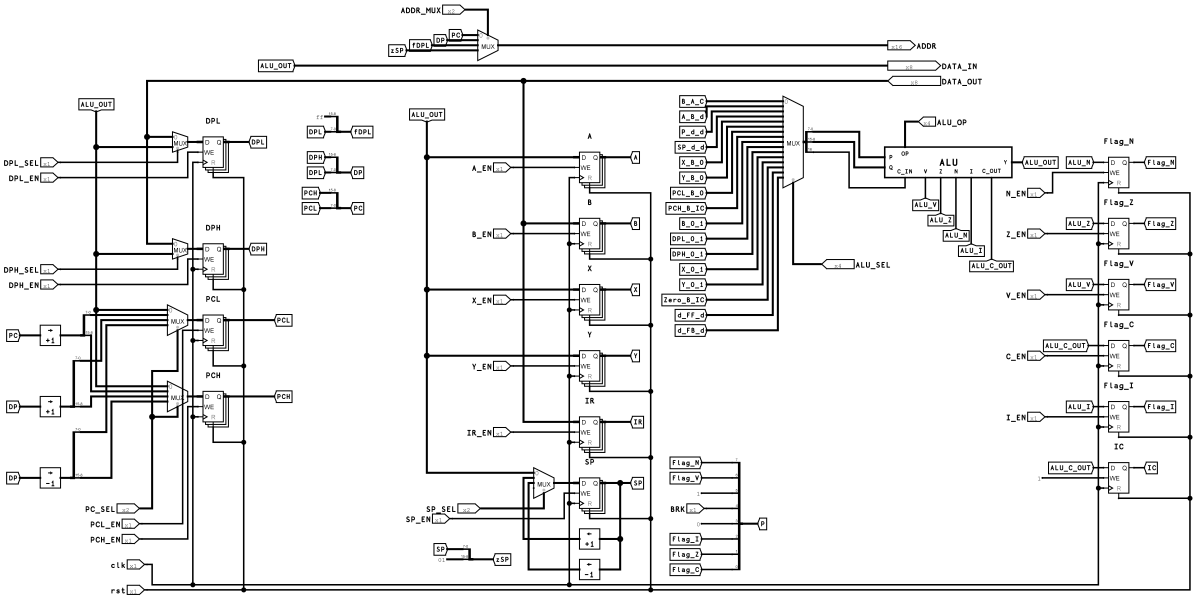
The overall CPU block diagram looks like:



The memory map is explained in the previous section, since it's straight forward so there are no detailed diagram for it.

Datapath

The diagram for data path is listed below (there are separate pdf file for this in case it's too small).

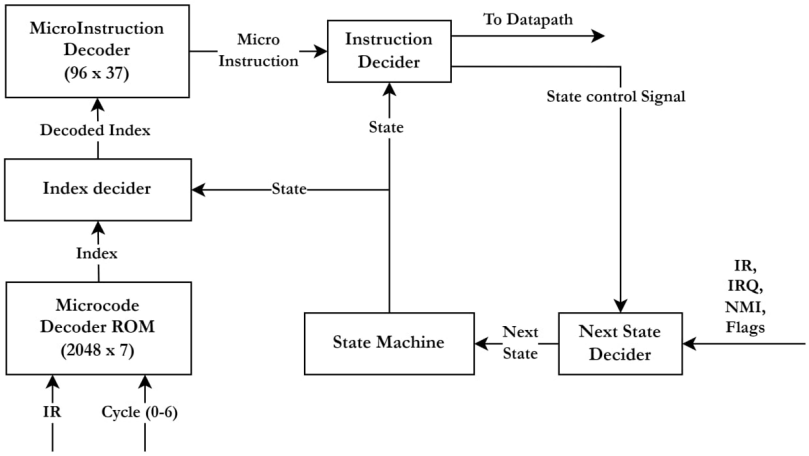


As you could see, all `SEL`, `EN`, `ALU_OP` and `BRK` are given by the decoder. All the register are mentioned in the previous section. This designed specifically for the needs of the microinstruction we have. Note the mux for ALU, the label `B_A_C` means that we pass the mux with a 17-bit logic of B register, A register, and the C flag. These labels are used frequently in the above diagram, otherwise it will be illegible.

The corresponding operation for ALU could be found in the excel sheet.

Decoder

For the decoder, PLA could be efficiently implemented using transistors on ASIC, but it's actually more efficient to just use the microinstruction approach like in today's x86 in FPGA case. The LUT (Look up table) is closely resemble the microcode decoder ROM and microinstruction decoder ROM so we will uses two ROM like the font ROM in Lab 7. It has diagram:



The decoder reads from IR and a cycle, which it keeps internally to track which cycle one instruction is in (since each instruction takes multiple steps). The microcode ROM will return a index to the Microinstruction ROM. Since we could probably executing a reset, IRQ, NMI logic, in this case we need the corresponding Reset, IRQ, NMI microinstruction ROM index instead of the index from IR and Cycle. So the Index Decider will decide the decoded index based on the state (which indicates whether we are in these Reset, IRQ, NMI), and gives this to Microinstruction Decoder.

The original 6502 uses an asynchronous memory, so with proper timing, it could read/write memory within one cycle. It's not the case for the synchronized BRAM in the FPGA. To do this, we split one cycle into two. For reading, we put the corresponding address on the address bus in first cycle, and then get the read value in next second cycle. For writing, we put the corresponding address on the address bus, enable writing, and put the write data on the data bus. This is all done in second cycle so have an idle first cycle for writing.

Our microinstruction merges the above said first cycle and second cycle control signal all in one 37 bit long data. For the first cycle, we extract only the bus operation control signal and leave other control signal as `noop` signal. For the second cycle, we just use the microinstruction (it's harmless to read the data again for reading). This is done by Instruction Decider.

In the microinstruction, there are 4 bit control signal designed specifically for branching and carrying operation. This signal together with the Flags signal let Next State Decider whether we go to a jump cycle or end the branching instruction. These 4 bit control signal also decides whether we should handle IRQ/NMI in next cycle.

Microcode/instruction generation

But how can we get these microcode and microinstruction? We might first start with some pseudo low level instructions. I will use the simple instruction `LDA #imm` as example. This instruction load the 8-bit literal value `#imm` into the A register. It completes in 2 cycles and is 2 bytes long. For its execution, we might want to write as:

`T1`		`LoadB(PC);`	`Continue;`
`T2`	`ToA(B,NZ);`	`LoadIR(PC);`	`End;`

The `T1`, `T2` means the cycle we are currently executing, we assume the correct instruction is already loaded in the IR before the first cycle. There are three columns that we deliberately divide the instruction into. First column means a ALU operation, second column means a memory operation, third column means a state control signal (the one in the decoder diagram). This way, we are aware not to write more than one pseudo instruction per slot, since we could not perform 2 ALU/memory operation in one cycle.

Back to the example, `LoadB(PC)` means load the `B` from the `PC` address, it has implied that `PC` will be incremented also uses its own incrementor shown on the data path. `Continue` signal to the decoder that we want to continue to the next cycle for this instruction. This moves the `#imm` that `PC` points to to `B`. In second cycle, `ToA(B, NZ)` do ALU operation that moves B to A, set the flags N and Z according to value of B. `LoadIR(PC)` will also load the instruction opcode for next instruction. `End` signal to the decoder that we switch the state back to `T1`, and go to NMI and IRQ handling if one presents.

Now we could write every instruction 6502 has using this method, and these are all listed in the attached excel sheet. Now we need to transform these pseudo instruction into actual control signals., these are listed in the Excel, and conversion is done automatically by few scripting code. Then we run a unique algorithm to see for each cycle, how many unique combination of these pseudo instructions. We makes these unique combination the microinstruction, and put them into the microinstruction array. For each instruction and its cycle, it's now associated with an index that corresponds to the microinstruction in the array. The opcode is 8 bit long, and there are 7 cycles max (so 3 bit to encode this information). Therefore, there are 11 bits long (2048 instruction-index possible). We make this 2048 entry long microcode table that associate each opcode-cycle to their microinstruction index. That's how we generate the microcode and microinstruction table.

PPU

Related information

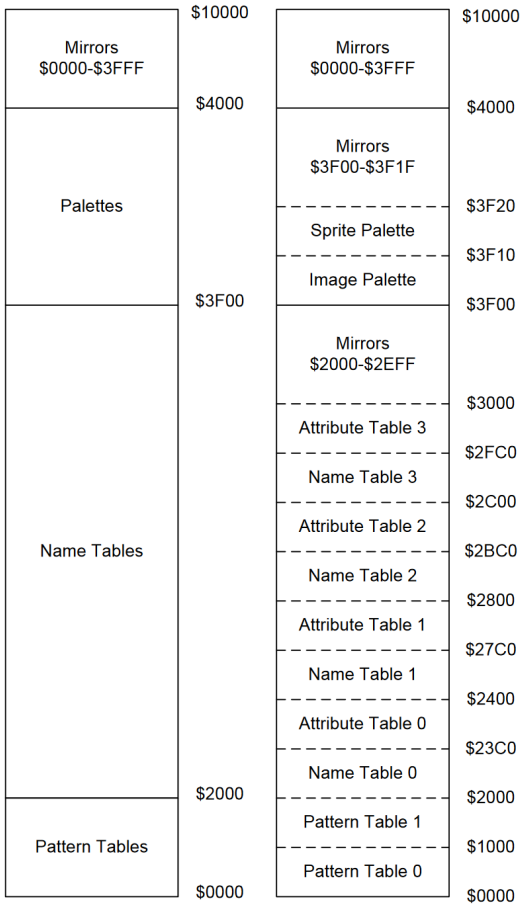
On NES, PPU will generate a 256x240 graphics, and outputting the corresponding TV signal to the TV monitor. Outputting

In our design, PPU is designed to run at 60Hz frame rate. Just like VGA, there are non-rendering cycles and ticks. The PPU has total of 341 ticks per scanline, and 261 scanlines. Therefore, the PPU should be run at a clock rate of 341x261x60 = 5.34006MHz.

Internally, PPU has a 64 color palette, these palette is fixed and could not be changed. To index the palette, there needs to be 6-bit index. The color palette is listed in page Palette. The color we used is

$00	$01	$02	$03	$04	$05	$06	$07	$08	$09	$0A	$0B	$0C	~~$0D~~	$0E	$0F
$10	$11	$12	$13	$14	$15	$16	$17	$18	$19	$1A	$1B	$1C	$1D	$1E	$1F
$20	$21	$22	$23	$24	$25	$26	$27	$28	$29	$2A	$2B	$2C	$2D	$2E	$2F
$30	$31	$32	$33	$34	$35	$36	$37	$38	$39	$3A	$3B	$3C	$3D	$3E	$3F

PPU has a memory map looks like below:



Each section will be explained below. In addition to the main memory, there are 256 bytes memory called sprite memory (OAM, Object Attribute Memory) used for storing sprite data.

Palette

there are Image Palette and Sprite Palette. Background and Sprite will be described later. Each palette has 16 8-bit values that are indices to the PPU internal palette. As we mentioned, we only need 6-bit index to the internal palette, so the 8-bit values here will have there 7th and 8th bit unused. The palette are listed like

Address	Purpose
$3F00	Universal background color
$3F01-$3F03	Background palette 0
$3F05-$3F07	Background palette 1
$3F09-$3F0B	Background palette 2
$3F0D-$3F0F	Background palette 3
$3F10	Mirror of universal background color
$3F11-$3F13	Sprite palette 0
$3F15-$3F17	Sprite palette 1
$3F19-$3F1B	Sprite palette 2
$3F1D-$3F1F	Sprite palette 3

> Notice that `$3F00, $3F04, $3F08...` are all the same (mirrored to `$3F00`).

To index into the palette (Background palette and sprite palette has 32 entries in total), we need a 5-bit index. These 5-bit are drawn from pattern table, attribute table, OAM respectively. It is determined as follows (we will call it palette number):

4	3-2	1-0
Is background (0) or Sprite (1)	Decided by attribute table or OAM	Decided by the pattern table

Pattern Table

Each pattern table contains 256 patterns. Each pattern is of 8x8 size. Pattern is used like:

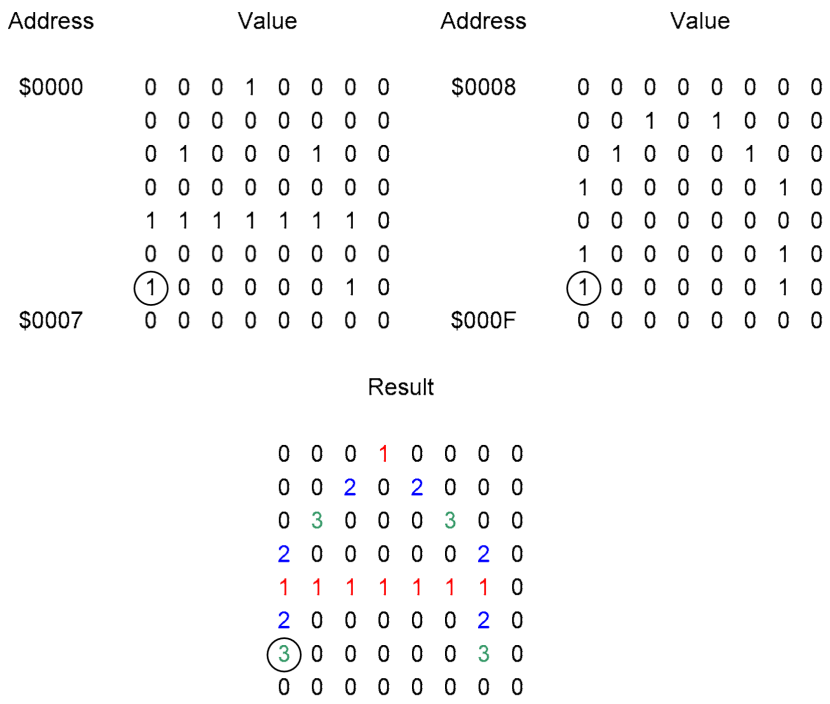


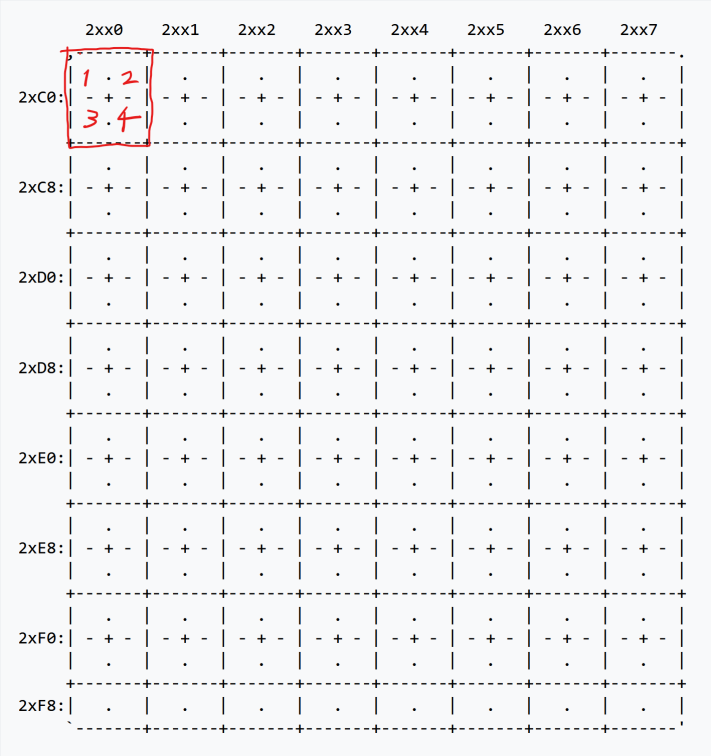
Figure 3-2. Pattern tables. Adapted from [7].

For each 8x8 pattern, 16 bytes of memory is used as shown above. The first 8 bytes decides the first index in the palette number, the second 8 bytes decides the second index in the palette number. Since each is 16 bytes, and there are 256 patterns, there are 256 x 16 bytes = 4KB total for each pattern table.

Name table and Attribute Table

As mentioned, the PPU renders a 256x240 size graphics. PPU actually uses the patterns to render the entire window. Since each pattern is of 8x8 size, that means we need 32x30 8-bit value for indexing to pattern table. These index are stored in pattern table. PPU will uses name table to find the correct patterns in the pattern table when rendering.

As mentioned, bit 3-2 of the palette number are decided by the attribute table, for the 32x30 name table, each one byte of data in attribute table is associated with 4 patterns (i.e. 4 name table values), so each pattern get 2 bit value for their palette number.



We could see that 20C0 should associate to the first name table's 4 patterns in the top corner, and so on. In each 2x2 area, we have

7-6	5-4	3-2	1-0
For pattern 4	For pattern 3	For pattern 2	For pattern 1

OAM

The OAM is used to store 64 sprites data. Each sprites take 4 bytes so the OAM is 256 bytes in total. For each sprite:

- Byte 0: top Y of the sprite.
- Byte 1: Pattern table index.
- Byte 2:

7	6	5	4-2	1-0
Flip Vertically	Flip Horizontally	front (1) of background	0	palette number

- Byte 3: left X of the sprite

PPU Control Register

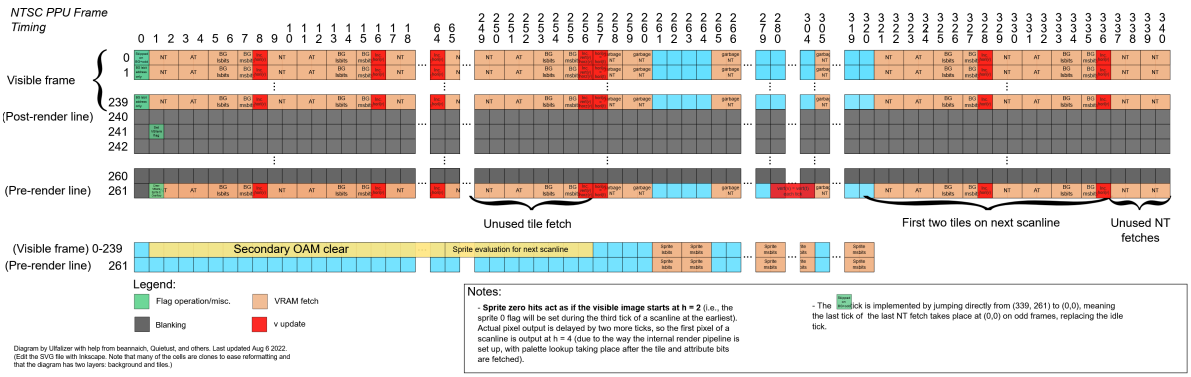
PPU has 8 control registers for CPU interaction. I have only implemented the `$2006` and `$2007` so I will only list them here.

- `$2006`: address of PPU memory. Since PPU memory is 16-bit, 6502 write the address twice to this register, first the 8-bit high byte, then the 8-bit low byte.
- `$2007`: PPU write/read data. After manipulating `$2006`, CPU could read the data in the specified address, or write to the specified address.

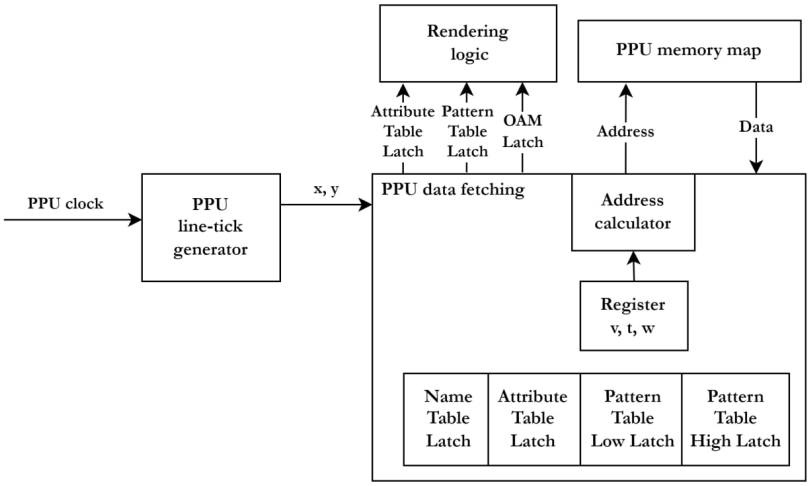
Implementation

Rendering

The NES dev specifies the [render timing] in the website.



Rendering and scrolling is a complicated mess, for the relevant information, one should consult [Nesdev PPU] directly. Here will only give an oversimplified explanation, the overall block diagram looks like



PPU renders at tick 1 to 256, scanline 0-239. PPU keeps 3 internal register for X, Y position tracking. They are register v, t and x. There are four index used for rendering: (1) coarse X (5-bit). (2) coarse Y (5-bit). (3) fine Y (3 bit). (4) fine X (3 bit). First three index is stored in the register v and t. The last index is stored in the register x directly. CPU interaction will only interfere the v register. t register is changed as PPU drawing the frame, v register is used to restore t register after each scanline and each frame.

PPU fetches data from its memory in 8-cycle period. In each period, using the mentioned four indices, it will decide which data table byte and attribute table byte to fetch. The address calculation is non-trivial and should refer to the website reference instead. Then it will fetch the corresponding pattern in pattern table. Since each pattern contains two 8-byte data, it will fetch two 1-byte data from the pattern. These pattern are then put into four 1-byte latch. This explains the orange part in the render timing picture.

The OAM is fetched in tick 260-320, I have not implemented the OAM in PPU yet. So this part is skipped.

Every 8 cycle, the coarse X is added by 1, to next 8 bytes of pixel data. The latches fetch are fed into the rendering logic where several shift registers are used to correctly find the current pixel data (the palette number), one should refer to PPU rendering directly and several other sources as it could not be explained within few pages.

At tick 256, the fine Y is incremented. At tick 257, the coarse X is reset to the one stored in the v. At pre-render line, the coarse Y and fine Y is reset to the one in the v. Note the increment is not trivial, it involves few wrapping around and strange PPU design decision. One should refer to PPU scrolling for more information.

PPU-VGA signal conversion

I have used the VGA-to-HDMI IP used in Lab 7. But as HDMI standards specifies, the lowest frequency needed for the HDMI clocks needs to be around ~25MHz, yet the PPU only outputs at around 5.34MHz. Since PPU frequency strictly tights to CPU frequency, we should not change the PPU-to-CPU frequency ratio (as that will make a lot of graphics rendering trick impossible), and we should not change CPU frequency (that will change how quick the game runs). Of course, it's impossible for us to cache entire one frame for PPU, as the memory needed is astronomical. (256x240x6 bits) We must find a better way to synchronize the PPU output and the VGA input.

First we notice that VGA has a size of 640x480, which could contain a 2x scaled up PPU graphic in the middle. VGA has 525 scanlines, PPU has 262 scanlines. So roughly speaking, if they both operates on 60Hz frame rate, one PPU scanline should roughly has same time as two VGA scanlines. So, if we could cache few PPU scanlines into a scanline buffer, and let VGA scanline lags behind by PPU scanline, we should be able to let VGA read the scanline buffer and output correct graphics. In my design, I uses 4 scanlines buffer, and let VGA starts at 521 (-4) scanline instead of 0 scanline.

The clocking wizard actually has some limitation, it could not produce 5.34MHz clock speed. This requires some VGA, PPU synchronization. Specifically, I tune the PPU slightly quicker then it should be, then at PPU scanline 0, tick 0, it will holds at that tick until VGA reaches scanline 521, tick 0 (considered to be the start of new VGA frame), this makes sure the PPU could correctly output the VGA graphics.

CPU interaction

PPU will generate an NMI signal every frame for CPU to update the graphics. The NMI is generated at scanline 241, tick 1 to 4 since PPU's clock is 3 times higher than CPU, 3 ticks of NMI signal generation makes sure CPU won't miss this NMI at its clock posedge.

PPU internally keeps a write strobe for 2 CPU write to the control register `$2006`. It always will increment this address for every CPU read and write to `$2007`. These access will perturb the v and t register, as in fact v and t register are used internally as address for read and writing the data. The manipulation to the v and t register is a non-trivial process one should refer to the PPU scrolling guide.

iNES file flashing

The first 16 bytes of iNES file is the header for relevant information. In Flag 6, it contain which mapper the game is using. In byte 7, it contains the size of game instruction rom (PRG-ROM) (in 32k bytes). In byte 8, it contains the size of game character rom (in 8k bytes). Then it has the PRG-ROM data followed by CHR-ROM data. The one I have used for my testing is the Super Mario Bros, which has a NROM (no cartridge swapping). I have written a scripting code that extract the PRG-ROM and CHR-ROM into .coe file for Vivado, which could be flashed just in Vivado editor.

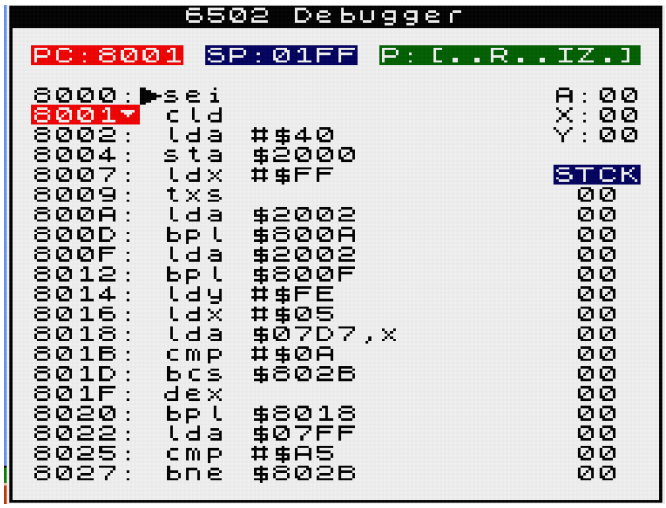
Module description

See module description pdf.

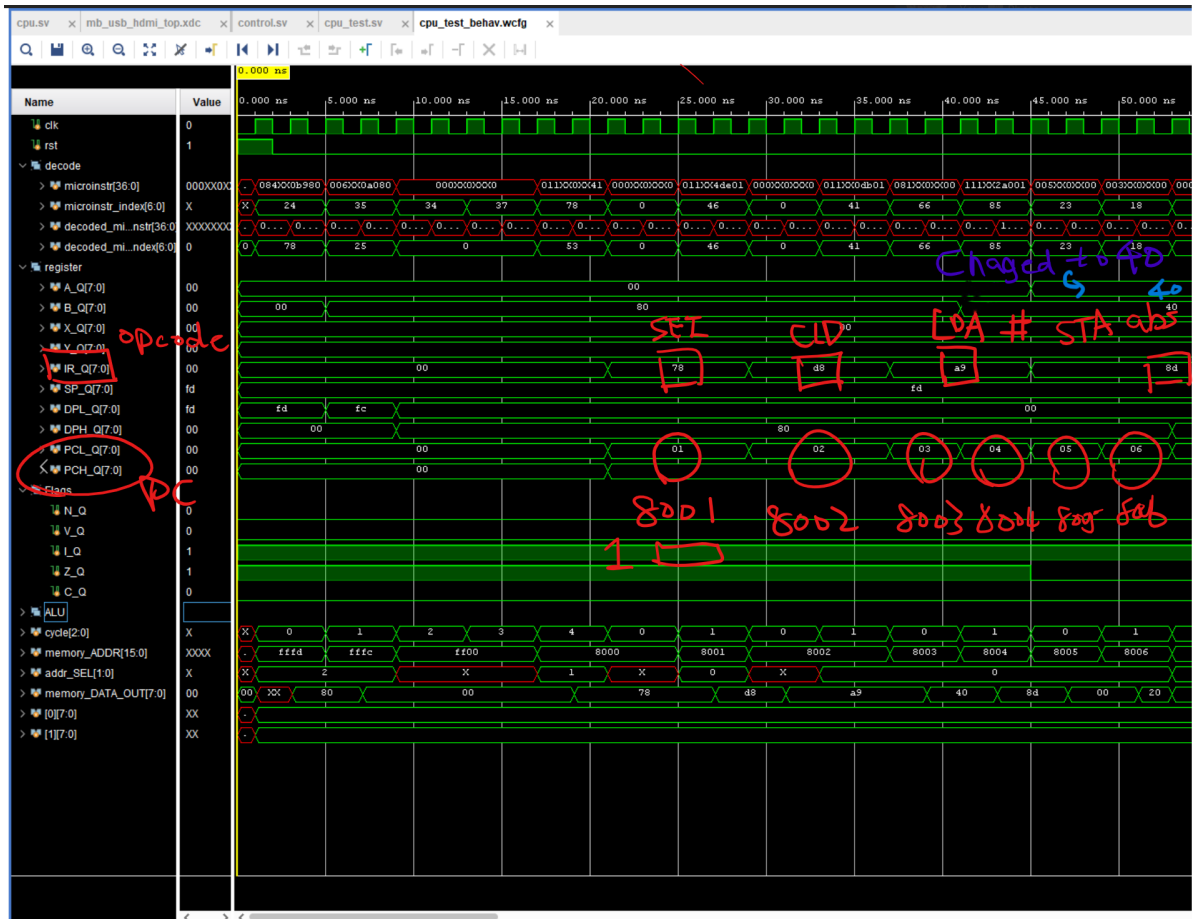
Simulation Trace

There is a simulation trace for running Super Mario Bros file directly.

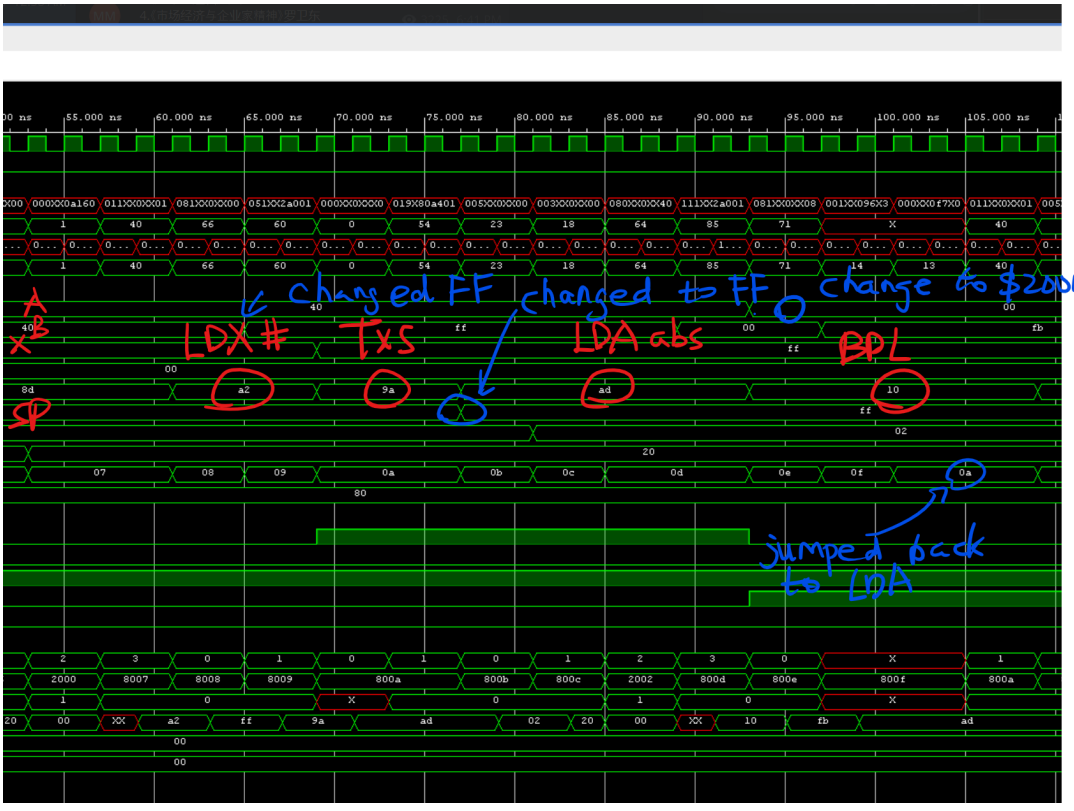
This is the 6502 debugger provided by iNES simulator.



The later traces show that CPU correctly simulate the instructions. (i.e exactly the same as software simulator).



the IR register shows the opcode for current instruction. The PC register shows the address of next fetch (so, for current fetched data, -1 from it). As we could see, we get `SEI` at address `8000` (8001 - 1). This set the interrupt (I) flag to 1 (no change, since the at boot up, the 6502 has interrupt disable flag on). Then we get `CLD`, which clears the decimal mode, in our case it's a `noop` since the NES 6502 doesn't support decimal mode at first place. Then we get `LDA #$40`, which will load hex value 40 into A register, which we could see in next trace, indeed get set to 40. Then we get `STA abs`, which does `STA #$2000` (sadly we couldn't see this in trace since it stored in block memory)



Then we have `LDX #`, and we see that X changes to `FF`, and then `TXS` and we see that X is moved to SP. and P becomes FF. Then we get `LDA abs`, which reads from `2002` and since we didn't implement the A get 00. The last we get `BPL`, this will jump when `N = 0`, since at `LDA abs`, we get `N = 0`, so this jumps back to `800A`, which is the `LDA abs` again. This traces thus show that CPU we created correct execute all its instructions.

Design Resources and Statistics

LUT	1100
DSP	0
Memory (BRAM)	14.5
Flip-Flop	702
Latches*	0
Frequency (MHz)	81.20
Static Power (W)	0.075
Dynamic Power (W)	0.244
Total Power (W)	0.319

note, the frequency part doesn't make too much since, as each part of the NES is running in its own speed set up by clock wizard.

Conclusion

In the project. I did research on the CPU, PPU and relevant part in the NES system. I have correctly implemented CPU and most part of the PPU part. I could run the Super Mario Bros and see the simulation trace that it runs correctly, and I could also running PPU outputting the Super Mario Bros character rom. Although I didn't finish the NES in time (which is hard for one person in 4 weeks), I might ask professor to borrow the board over the weekend (or buy one) to finish the NES.