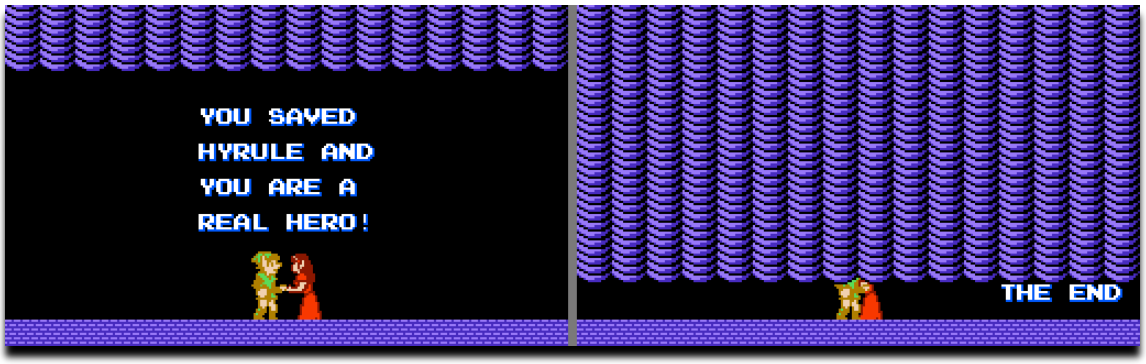
Emulating the Nintendo Entertainment System (NES)



Written by Rupert Shuttleworth for

COMP4121 2007s2

(taught by Aleksandar Ignjatovic)

Overall layout and design

The ball gets rolling with:

```
  #include "nes.h"

  NES nes = nes_init(filename, width, height);
  nes_run(nes);
  nes_destroy(nes);
```

A NES itself is composed of:

```
struct nes {
  Memory cpuMemory;
  Memory ppuMemory;
  Memory objectAttributeMemory;

  CPU cpu;
  PPU ppu;
  APU apu;

  Cartridge cartridge;

  Interrupts interrupts;

  MMU mmu;

  GUI gui;

  Joypad joypads[NES_NUM_JOYPADS];
};
```

The CPU (Central Processing Unit) and PPU (Picture Processing Unit) each have their own address space, the reason their memory is at the top level here and not contained within the CPU or PPU objects is that the Cartridge (and its Memory Mapper Unit - MMU) needs to setup memory access callbacks all over their address space.

There is also a separate area of memory for sprite attributes, officially called "object attribute memory". The CPU writes to this and the PPU reads from it to determine sprite details like position, colour, shape, etc. It sits at the top level to represent that it is "shared" between the CPU and PPU.

The CPU, PPU and APU (Audio Processing Unit) also have their own objects to keep track of internal data, such as storing register contents as well as any administration details such as what the current scanline to render is.

The Interrupts object is a top level object because many devices are free to generate interrupts, which are then handled by the CPU (unless it is currently ignoring interrupts). There are three general types of interrupts

- NMI - non maskable interrupts, interrupts which the CPU cannot "mask away", it must handle them
- IRQ - interrupt requests, interrupts which the CPU is free to ignore
- RESET - the initial power on / reset interrupt generated when you hit the power switch

Each of these has their own specific interrupt handlers living in the cartridge ROM, which programmers must supply.

The CPU can also generate "fake" interrupts by a programmer giving it the BRK opcode, which are treated as IRQ interrupts and trap to the IRQ handler.

The GUI is used for rendering output and interacting with the user.

The Joypads are used for storing current joypad state, such as what the last button state requested was. They talk to the GUI to actually check this.

In general, the CPU repeatedly checks whether there is a pending interrupt it could handle, otherwise it tries to perform the next program instruction. During execution of the program instruction, the PPU and APU also make progress, according to the number of cycles the CPU takes to perform the operation. In the real NES, they are running in parallel, here we simulate that by stepping them in parallel.

Emulating the CPU (Central Processing Unit)

The NES uses a modified MOS 6502 CPU core, a processor that was heavily used in the 70s and 80s, showing up in early Apple machines and other early videogame consoles like the Atari 2600 and the Commodore 64. It has only a handful of registers but a large number of instructions which can be used in over a dozen different addressing modes. One addressing mode, "zero page", takes a one byte address instead of a two byte address as an operand, limiting the address to reside in the first page of accessible memory (the "zero page"). In a sense this allowed the CPU to have 255 different zero page "registers" because the instructions were quick to execute, although they were still slower to access than genuine registers.

Nintendo used the 6502 over 8 years after its original development, and retrofitted its memory addressing system to map memory addresses to different hardware devices (discussed in the next section.) Since the standard 6502 CPU was widely used, it has a lot of fans who have written lovely technical documents detailing exactly how it should behave. This made emulating it rather easy. The main challenge was setting up the memory mapped IO, and also tracking down particular changes Nintendo made to the standard 6502 chip design which weren't covered by the more general documentation, such as changing the way addresses wrapped with indirect jump instructions.

For a full reference on the exact op codes and addressing modes please see the appendix, however in general, the CPU fetches an instruction, fetches the byte after the instruction (which it might throw away if the instruction didn't need it), and possibly fetches even more bytes depending on the addressing mode. Eventually it has worked out the final address of the data which is to be acted on, and it calls a function to carry out that modification, passing it the address of the data to use. Many instructions update status flags which keep track of things such as, was the last number negative, did the last number overflow, etc. After an instruction completes, programmers can use additional instructions to query the status flags set by the previous instruction. This is usually done with branch instructions, such as the “BEQ” instruction which will branch if the status flag records that the last operation resulted in a number equalling 0.

The CPU emulation was one of the first things to get written, and I spent a long time trying to get the timing correct relative to the other devices (ie. picture and audio processing units). On the real hardware, all three devices run in parallel so my first instinct was to run them in separate threads, centered around a memory bus abstraction where they could make read/write requests to the address and control buses and then read/write from the data bus. Each thread had to gain exclusive access to the bus before trying to modify it and synchronisation was enforced by having barriers after address/control bus changes and after data bus changes to make sure that each device had witnessed the current values.

I thought this would be neat because it was close to how the real hardware worked. However it was too slow. I wrote a prototype using pthreads that used semaphores and barriers to enforce the synchronisation and at most I was getting ~200,000 bus operations per second, which is an order of magnitude less than the real NES hardware.

So I went back to a sequential implementation, however I had to make sure that the relative time spent on each device was accurate. I read many articles and suggestions on the best way to do this, a lot of them involved the CPU keeping an exact track of how many cycles the current operation had taken, and then passing that number to the audio and picture processing units to see whether, in that amount of time, they could have done something themselves. This seemed messy and I didn't like it.

Another suggestion I read was on using memory accesses to synchronise hardware in a transparent way without having to pass around a CPU cycle count variable to other devices. In my sequential program I implemented this by writing wrapper functions which guard around access to the CPU memory, and whenever the CPU wants to read or write to memory it has to go through the wrapper functions. These functions do indeed pass the byte to be written on to the real CPU memory (or return the byte to be read from the real memory), but they also allows the audio and picture processing units to advance by calling *ppu_step* and *apu_step*.

The problem then became working out, for any given memory access made by the CPU, how many cycles the PPU or APU could have advanced in that time, and then calling *ppu_step* or *apu_step* that many times from the wrapper. For an NTSC NES this turns out to be 3 steps for every memory access made by the CPU, although this would change slightly for a PAL NES.

As a side node, the original 6502 really does fetch a second byte for instructions, even if the instructions only needed one byte. The important thing in emulating this was that the timing was correct, not that a dummy byte was fetched and thrown away. So I added a function *nes_cpuCycled* which tells the NES that the CPU just spent a cycle "in some way", simply in order to allow the other devices to advance the amount of time the pointless memory access would have taken, without actually bothering to fetch a throwaway byte.

The *nes_cpuCycled* function ended up being pretty useful because some other instructions used pipelined memory access, which meant that operations which might cost 1 cycle on their own, like incrementing a stack pointer, didn't necessary cost 2 cycles when done twice, as the first increment could happen in parallel with another operation (like reading the byte from the stack). The "waste a cycle" function allowed functions to determine for themselves how data would be pipelined, if they needed to, otherwise they could use the wrapper functions for reading and writing CPU memory to allow other devices to automagically advance.

Interestingly, the real 6502 CPU actually has the ability to create "undocumented" op codes, in a bizarre way. OP codes are 8 bits, giving 256 possible combinations. The 8 bits can be viewed as 4 high bits and 4 low bits used as vertical and horizontal indexes into

a 2D matrix of "merged" operations, where both the high bits and the low bits have an effect on the final operation that is performed (see appendix for full chart.) Only 151 of these combinations result in documented, official op codes, leaving 105 combinations free for "experiments" where, for example, various operations are run under addressing modes that they aren't supposed to work with. Operations like "TNP" (tripple-NOP, three no operations) and "KIL" (which stops the program counter) have been "invented" by people experimenting with illegal opcodes. I haven't attempted to emulate this process as it is pure voodoo, instead I only emulate the genuine opcodes.

Using callbacks to emulate memory mapped IO and mirroring within the CPU and PPU address spaces

The CPU and PPU use memory mapped IO to talk to different devices, so that when the CPU attempts to write to a memory address in CPU memory space it might really trigger an effect in the APU, or PPU, or Joypads, or ROM data on the cartridge, etc. And when the CPU attempts to read from certain addresses, different devices might intercept this and return data from their own memory space. For example, if the CPU attempts to read from address 0x4016, this actually triggers a read request on the first Joypad controller.

Similarly, when the PPU (again, Picture Processing Unit) makes read requests on certain memory spaces, it might actually trigger the ROM on the cartridge to return data instead of pulling the data out of the real PPU memory RAM.

There are also many areas of CPU and PPU address space which are mirrored, so that programmers have many different ways to access the same data. This saves a few operations in having to update absolute memory addresses, instead letting addresses wrap around and mirror previous addresses. This means that when a write request is made on address X, it might also affect address X+8, address X+16, address X+24, etc, depending on how the mirroring is determined.

To model all of these, I created a Memory ADT which contains enough real RAM to store an arbitrary amount of data, but also has, for each individual address, the ability to add either a read callback function or a write callback function (or both) so that when read or write requests are made, instead of being written to RAM (ie. written to an array of bytes in the Memory ADT), they instead trigger an arbitrary function call to happen. That function call is given a reference to the address wanting to be read/written to, as well as the actual data being written if it was a write request. It is also given a reference to a "NES" object, a kind of God object that can allow changes to be made to many areas of the hardware. For example, given a NES object, one can call *nes_getCPUMemory(nes)* to grab a reference to the CPU's memory space.

Similarly, for mirroring of addresses, one can simply setup callbacks in the mirror range that redirect requests back to the "base" addresses being mirrored. One nice thing with this is that the redirection can actually trigger yet another callback.

For example, in the CPU memory address space, writing to addresses between 0x2000 to 0x2007 affects registers in the PPU.

For example writing to CPU memory address 0x2000 affects the PPU Control Register, which is an independent register sitting in the PPU. By independent I mean, it is not the case that the PPU reads CPU memory address 0x2000 and treats that as a "register", instead the PPU has a separate, (fast) register that just gets updated by writes to CPU memory address 0x2000 but is otherwise read by the PPU independently.

The setup for the 0x2000 mapping in the CPU memory space is:

```
  memory_setWriteCallback(memory, CPU_PPU_CONTROL_REGISTER_ADDRESS,
&cpuMemory_ppuControlRegister_writer);
```

where cpuMemory_ppuControlRegister_writer is a function defined as:

```
static void cpuMemory_ppuControlRegister_writer(NES nes, Address address, Byte data) {
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  ppu_setControlRegister(ppu, data);
}
```

So you can see that, when a write request is made, the callback function passes it on to the PPU ADT to update the status register.

Additionally, the 0x2000... 0x2007 address range is actually mirrored repeatedly (every 8 bytes) in CPU memory space, until address 0x3FFF. So one has to also setup mirror callbacks for those, eg. mirrors for writing:

```
  for (address=CPU_PPU_MIRROR_FIRST_ADDRESS; address <= CPU_PPU_MIRROR_LAST_ADDRESS;
address++) {
    memory_setWriteCallback(memory, address, &cpuMemory_ppuMirror_writer);
```

```
  }
```

where cpuMemory_ppuMirror_writer is a function defined as:

```
static Address cpuMemory_ppuMirror_getLowestAddress(Address address) {

  while(address > CPU_GENUINE_PPU_LAST_ADDRESS) {
    address -= CPU_PPU_MIRRORED_SIZE;
  }

  assert(address <= CPU_GENUINE_PPU_LAST_ADDRESS);

  return address;
}

static void cpuMemory_ppuMirror_writer(NES nes, Address address, Byte data) {
  address = cpuMemory_ppuMirror_getLowestAddress(address);

  Memory cpuMemory = nes_getCPUMemory(nes);
  assert(cpuMemory != NULL);

  memory_write_callback(nes, cpuMemory, address, data);
}
```

So, when a write request is made to an address in this range, the callback function rolls the address back to the genuine address, then triggers a memory write for that location.

memory_write_callback is defined (in the Memory ADT, hence memory_ prefix) as:

```
void memory_write_callback(NES nes, Memory memory, Address address, Byte data) {
  assert(memory != NULL);

  if (memory->writeCallbacks[address] == NULL) {
    memory_write_direct(memory, address, data);
  } else {
    memory->writeCallbacks[address](nes, address, data);
  }

}
```

So *memory_write_direct* will write to the raw RAM data if there is no callback function setup, otherwise the callback function is triggered and it allowed to do whatever it wants.

This allows a write to the mirror callback, to redirect to the original address being mirrored, and then (if there is a callback setup there), to trigger yet another callback to trigger. It is really nice!

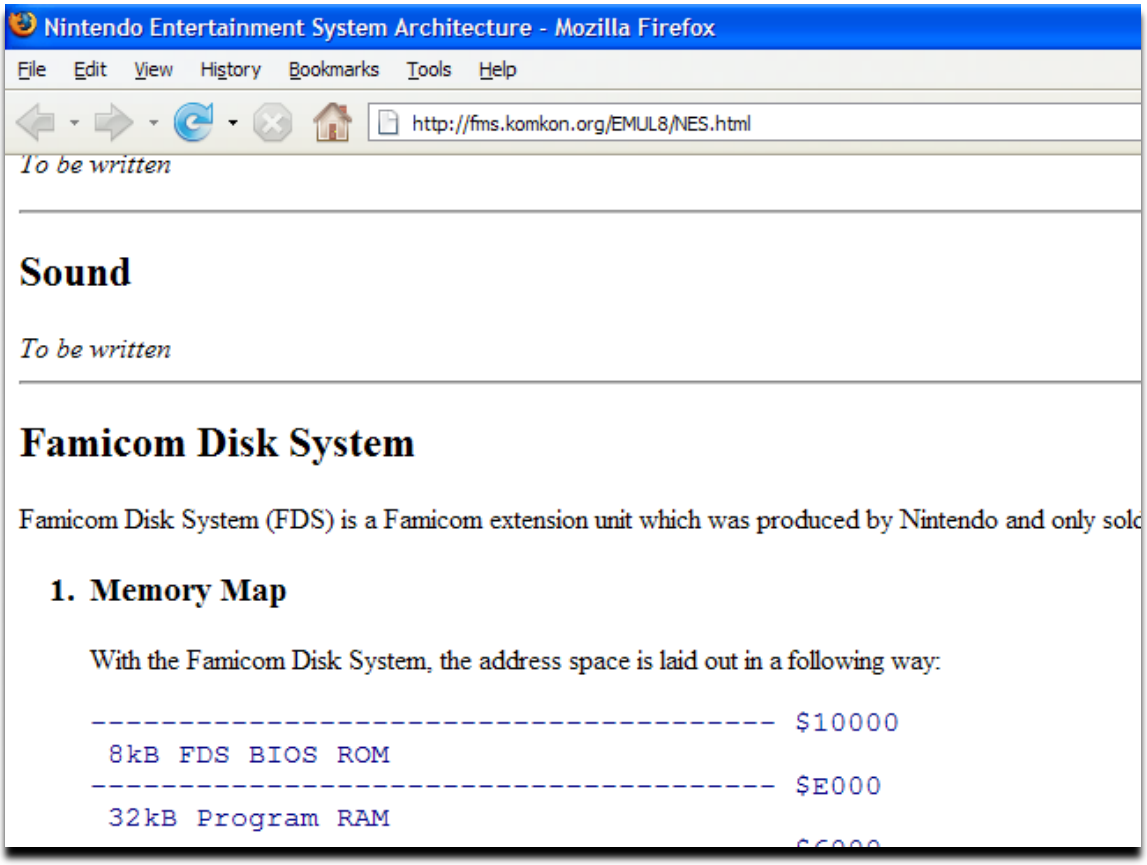
This whole process complicated the design, and made testing a pain because what were originally independent parts of the program suddenly had access to (and relied upon) many other modules, eg. the Memory ADT suddenly had a NES ADT reference to deal with, while the NES ADT itself had several Memory ADTs to deal with. This made it really hard to attempt to test modules by themselves. However I think introducing a God object was a necessary sacrifice to make because that is how the real hardware works, with a seemingly innocent read or write request able to have great repercussions on other parts of the system. This approach of using callbacks also let memory addresses "look after themselves" after the initial setup phase was over, allowing me to avoid a huge memory checker which would have looked like

```
if (address = ... ) {

} else if (address >= ... && address <= ...) {

} else if (address >= ... && address <= ...) {

}

...

// 400 lines of else if later...
```

I was my pleasure to be able to avoid writing something like that.

Emulating the APU (Audio Processing Unit)

I actually don't know much about how the sound hardware works. I do know about how the CPU would communicate with the APU (through memory addresses starting at 0x4000 in CPU memory address space, which map to registers in the APU). This would be the same process as how the CPU communicates with the PPU so it is easy enough to implement by setting up additional callbacks in the CPU memory space. However, what actually happens in the APU once the registers are updated is a mystery to me. Many of the documents I read didn't cover sound at all, or had a section for sound that just contained a comment like "coming soon", so emulating sound was a low priority for me. However there is still a bit of information available on it so I might try to emulate this at a later date.



Emulating Cartridge data and their effects on CPU/PPU operation

Cartridges have a series of (1 or more) "program banks" (to store program code), as well as a series of (0 or more) "character banks" (to store the patterns for tiles and sprites). Program banks are each 16 kilobytes in size, and character banks are each 8 kilobytes in size. The CPU can access two program banks at any given time, by reading and writing from addresses starting at 0x8000 (for the first bank) and 0xC000 (for the second bank) in the CPU memory address space.

The PPU can address up two character banks at a time, which show up at 0x0000 and 0x1000 in the PPU address space. These are arranged into "tiles" (series of 16 bytes), which represent (aligned) 8x8 pixel areas of screen real estate, giving 2 bits of colour data for each pixel. The pattern tiles collectively form "pattern tables" of possible background shapes to render on the screen.

Each pattern tile stores not only colour information but also shape information, since the colour could be transparent. The 16 bytes used for each 8x8 tile are arranged so that the first 8 bytes form an 8x8 grid of bits:

```
              Bits in byte
Address 00: 00 00 00 00 00 00 00 00
Address 01: 00 00 00 00 00 00 00 00
Address 02: 00 10 00 00 00 00 00 00
Address 03: 00 10 10 00 00 00 00 00
Address 04: 00 10 10 10 00 00 00 00
Address 05: 00 10 10 10 10 00 00 00
Address 06: 00 10 10 10 10 10 00 00
Address 07: 00 10 10 10 10 10 10 00
```

This might look like 16 bits per byte instead of 8 bits per byte. However, only the first bit is the one stored in the actual byte. The second bit here is always 0, and is just included to demonstrate that the first 8 bytes affect the upper bit of a final, 2 bit colour index.

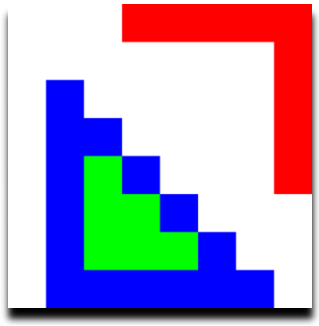
For the second series of 8 bytes, the lower bit of the 2 bit colour index is stored in the actual byte. The upper bit in the diagram below is always 0, as it is taken from the first 8 bytes.

```
              Bits in byte
Address 08: 00 00 00 01 01 01 01 01
Address 09: 00 00 00 00 00 00 00 01
Address 10: 00 01 00 00 00 00 00 01
Address 11: 00 01 01 00 00 00 00 01
Address 12: 00 01 00 01 00 00 00 01
Address 13: 00 01 00 00 01 00 00 00
Address 14: 00 01 00 00 00 01 00 00
Address 15: 00 01 01 01 01 01 01 00
```

The final colour index is formed by merging the two bits

```
               Local X
Local X 00: 00 00 00 01 01 01 01 01
Local X 01: 00 00 00 00 00 00 00 01
Local X 02: 00 11 00 00 00 00 00 01
Local X 03: 00 11 11 00 00 00 00 01
Local X 04: 00 11 10 11 00 00 00 01
Local X 05: 00 11 10 10 11 00 00 00
Local X 06: 00 11 10 10 10 11 00 00
Local X 07: 00 11 11 11 11 11 11 00
```

If we assumed that (details later), 00 corresponded to "transparent", 01 corresponded to "Red", 10 corresponded to "Green", 11 corresponded to "Blue", then the final image in this case is:



There is a separate area in PPU memory for "nametables", which are a series of bytes which contain indexes into the pattern tables to select which pattern to use. All the bytes in a nametable collectively form the layout of an entire screens worth of background data. Since each pattern tile is 8x8 pixels, and the screen resolution is 256x240 pixels, there are a total of 32 horizontal tiles and 30 vertical nametable tiles. Collectively there are 32x30 = 960 tiles per screen. A nametable contains 960 bytes of data to store, for each tile location, the index of the pattern tile to use from the pattern tables.

Each nametable is actually 1024 bytes, so the remaining space (1024-960 = 64 bytes) stores extra colour information for each of those proceeding tile locations, called attribute data. Remember that at this point, the pattern tiles themselves are only giving us 2 bits of colour for each pixel, which would result in a pretty crummy final image.

Now, there are only 64 bytes left in the nametable, but there are 960 nametable tiles, so each byte of the 64 attribute data bytes actually ends up affecting multiple nametable tiles. The screen as a whole is divided into 64, 32x32 pixel tiles, called attribute tiles, which each take 1 byte of data. This would actually contain enough attribute information for a 256x256 pixel output resolution, although the NES only uses 256x240 pixels worth.

Each attribute byte affects 32x32 screen pixels, and each nametable tile is 8x8 pixels, so each attribute tile ends up affecting 16 nametable tiles (32x32 / 8x8). However it does this in a very annoying way. Remember that each nametable tile references a pattern tile to use for that section of the screen, and each pattern tile stores data for 8x8 pixels. Since the screen has a resolution of 256x240 pixels, that means there are 32 horizontal nametable tiles and 30 vertical nametable tiles.

Each nametable tile is given a number starting from 0, from the left to right on the screen, and then increasing from top to bottom, so that pattern tiles in the same horizontal position but with a vertical difference of one will have pattern tile ids that differ by 32 (as there are 32 pattern tiles per row, since rows have 256 pixels and each tile has 8 pixels).

Since each nametable tile takes up 1 byte of data in the nametable (to store the index of the pattern tile to use for that position), the byte for each nametable tile can be found by treating its nametable tile id as a byte offset from the start of the nametable.

For example, if the screen resolution was 8x8, 8x8 tiles = 64x64 pixels, it would be formed of tiles with these numbers:

```
00 01 02 03 04 05 06 07
08 09 10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 24 25 27 28 29 30 31
...
56 57 58 59 60 61 62 63
```

And, for each of these tiles, the pattern index to use them would be stored as sequential bytes in the name table memory

eg. if the name table memory contained bytes with values:

```
15 12 00 13 15 ...
```

That would be that, nametable tile 00 used pattern tile 15, nametable tile 01 used pattern tile 12, nametable tile 02 used pattern tile 00, and so on.

The NES screen resolution is actually 256x240 pixels so the 64x64 example is just a toy example that I could fit on this page, however the 256x240 resolution uses the same process.

Now, back to the attribute bytes. Remember that each attribute tile affects 32x32 pixels, and each pattern tile is 8x8 pixels, so each attribute tile affects 16 pattern tiles in total. But the way these are laid out on the screen uses a different numbering system to the way that nametable tiles are numbered

If we go back to the previous example, which used a 64x64 screen resolution:

```
00 01 02 03 04 05 06 07
08 09 10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 24 25 27 28 29 30 31
...
56 57 58 59 60 61 62 63
```

Let's say that, even in this tiny 64x64 screen world, each attribute tiles still affected 32x32 tiles areas of the screen. Then there would be 4 attribute tiles for this screen, numbered in the same way as the nametable tiles were. The numbers given to each attribute tile would be:

```
0 1
2 3
```

And, again, each attribute tile id could be viewed as a byte index from the start of the attribute table (stored at the end of the nametable), to find the attribute data byte for that attribute tile.

In this world, attribute tile 0 would affect these nametable tiles:

```
00 01 02 03 ...
08 09 10 11 ...
16 17 18 19 ...
24 24 25 27 ...
..............
```

and attribute tile 1 would affect these nametable tiles:

```
... 04 05 06 07
... 12 13 14 15
... 20 21 22 23
... 28 29 30 31
..............
```

From the attribute tiles perspective, the nametable tiles it affected would then be renumbered.

So, for attribute tile 0, which really effects these nametable tiles:

```
00 01 02 03 ...
08 09 10 11 ...
16 17 18 19 ...
24 24 25 27 ...
..............
```

The nametable tiles would now be renumbered as:

```
00 01 04 05 ...
02 03 06 07 ...
08 09 12 13 ...
10 11 14 15 ...
..............
```

Now, remember that each attribute tile has 1 byte of data it uses to affect the nametable tiles it affects. The 32x32 pixel attribute tile, which affects 4x4 = 16 nametable tiles, is now split into 4, 16x16 sub-attribute-tiles, which each affect 2x2 = 4 nametable tiles.

Continuing on the previous example, we have the real nametable tiles

```
00 01 02 03 ...
08 09 10 11 ...
16 17 18 19 ...
24 24 25 27 ...
..............
```

Which are locally represented with new ids from 0 to 5 from the attribute tiles perspective

```
00 01 04 05 ...
02 03 06 07 ...
08 09 12 13 ...
10 11 14 15 ...
..............
```

And now the attribute tile has been split up into 4 sections, which we number as section:

```
00 00 11 11 ...
00 00 11 11 ...
22 22 33 33 ...
22 22 33 33 ...
..............
```

Now, remember again that an attribute tile has 1 byte of data it uses to affect the nametable tiles in its range. A byte has 8 bits, and we have now split the attribute tile into 4 sub-attribute-tiles, each of which affects 4 nametable tiles.

Each of these sub-attribute-tiles is given 8/4 = 2 bits of colour data to affect those 4 nametable tiles. The bits of the nametable byte are arranged as 33221100, so that the first two bits represent colour data for attribute-sub-tile 3, which affect (renumbered) nametable tiles 12, 13, 14 and 15.

The combined colour index for any given background pixel on the screen is found by looking up which nametable tile affects that pixel, then from the nametable tile, fetching the pattern tile for that nametable. Once we have the pattern tile, that gives us the lower 2 bits for any pixel covered by it. The upper two bits are taken from the attribute table. Together this forms a number with a 4 bit range, so 2^4 = 16 possible values.

Once we have this number, we use it as an index into image palette bytes stored at the end of the PPU memory space.

There are two image palettes, one for background images and one for sprite images.

Each palette has 16 bytes of data (naturally, since we have just generated a 4 bit index). Each byte contains an index into the global system palette to find the real colour to use. To find out which image palette byte to read, we use the 4 bit number given to us by combining the pattern and attribute data as an byte offset from 0..15 into the 16 byte image palette.

The byte at that address could theoretically contain 2^8 = 256 values, however the global system palette only contains 64 colours so only 4 bits of the byte are used.

Although each image palette could theoretically reference 16 of these colours, mirroring is used within the image palette so that every 4th byte is a copy of the 0th byte (used for determining background/transparency colours). Therefore each image palette only stores 13 genuine colours (4+3+3+3). Since there are two palettes, one for background images, one for sprites, this would let us (in theory) have 26 different colours on screen at once.

However, each 4th byte of the sprite palette (including the 0th byte) actually mirrors back to every 4th byte of the background palette (which themselves mirror back to the first byte of the background palette), so there is an overlap of (at least) one colour between the background and sprite palettes. This means that at most we could have 13 + 13 - 1 = 25 unique colours on screen at once. I've read that there are cartridge memory mappers which can increase this (more on memory mappers later) but haven't encountered one yet.

Let's return to the cartridge. Program banks can still contain image-related data (such as nametable layouts and image palettes), but in this case, the CPU will have to shift that data from the CPU memory space to the PPU memory space. It could do this by writing a PPU memory address to CPU memory address 0x2006 which sets the current address in PPU memory land, and then by writing bytes one at a time to CPU address 0x2007. Writes to both of these addresses are mapped to PPU registers and using them the PPU could transfer the data into the (otherwise private, from the CPUs perspective) PPU memory space.

This would be (reasonably) quick as the PPU will auto increment the current address stored in 0x2006 when writes occur. So the CPU could set the starting address once, then perform a series of alternate read (from CPU memory) / write (to PPU memory) operations.

However, the PPU has a separate section of memory for storing sprite attributes (x and y positions of sprites on the screen, pattern and colour data for sprites, recording whether a sprite should be flipped horizontally or vertically, etc...). This section contains 64 sprites, and each sprite has 4 bytes of associated data so in total it is a separate 256 byte section of memory. The CPU cannot access this sprite data by writing to 0x2006/0x2007 as it is not directly part of the PPU address space.

Instead the CPU has to set a sprite address into register 0x2003 and then write sprite data into register 0x2004 to affect the sprite memory, which are mapped to the sprite memory address space. The problem with this is that writes to 0x2004 do not automatically increment the current sprite address stored in 0x2003 (unlike writes to 0x2007, which do increment the current PPU memory address stored in 0x2006.)

So if the CPU wanted to fill up the entire 256 byte bank of sprite information by writing to 0x2003 and 0x2004, it would need to perform at least 4 logical operations per byte:

- reading (or incrementing) a sprite address index
- writing the sprite address index to 0x2003
- reading a byte of sprite data from somewhere in local CPU memory
- writing that byte to 0x2004

To fill up the entire 256 bytes of sprite memory this way would take thousands of CPU cycles, so Nintendo added a second way to fill up the sprite data, which involves a single write to CPU memory address 0x4014. The data written represents the starting address (divided by 256) in CPU memory space of 256 bytes worth of sprite data. This is mapped to a DMA device which will read sequential bytes from this address and write them into the sprite memory data starting from address 0, 1, 2, 3.. without any further involvement from the CPU. This still occupies the memory bus channels so prevents the CPU from doing anything while the transfer is happening, but the transfer only ends up taking hundreds of CPU cycles worth of operations rather than thousands of cycles worth.

Cartridge memory mappers

Each cartridge contains a memory mapper which determines which program and character banks are currently mapped into the CPU and PPU address space, as well as determining how mirroring is used in the PPU address space.

Only two program banks are visible to the CPU at a time (32 kilobytes worth of data), and two character banks are visible to the PPU at time (16 kilobytes worth of data), and the memory mapper in the cartridge is in charge of determining which program and character banks are currently active.

Currently I have only implemented the original NROM mapper (referred to as memory mapper 0) which is fairly restrictive. It allows for only 1 or 2 program banks on the cartridge, and if there is only 1 program bank that bank is mirrored in the CPU memory space so even though the CPU thinks it can see two banks it is really just seeing two copies of one bank.

The NROM mapper also allows for 1, 8 kilobyte character bank, which (if present) takes up both pattern table 0 and pattern table 1 in the PPU memory address space in their entirety as they address 4 kilobytes each. If no character bank is provided, then the two pattern tables in PPU memory address space are treated as RAM and not ROM, and the CPU will have to fill them manually by writing to addresses 0x2006 and 0x2007 in PPU memory space as discussed earlier.

In terms of the actual code, this is currently handled by an array of function references which are used to initialise specific memory mappings between cartridge ROM and CPU/PPU address space. Memory mappers can add callbacks to both CPU and PPU memory address space, which can intercept reads and writes made by the PPU or CPU and redirect them to the cartridge ROM (or redirect them nowhere, in the case of write attempts made to cartridge ROM.)

My original solution to this was different, and involved manually copying bytes between cartridge ROM and CPU/PPU memory address space to fill up the entire 32 kilobytes of addressable program code and 8 kilobytes of addressable character code. However I ended up shifting everything to callbacks because it is much more elegant and more powerful. Some of the more exotic memory mappers (like MMC1 and MMC3) contain more data on the ROM than can be accessed by the CPU/PPU at any given time, and want to be able to dynamically change the active banks, sometimes thousands of times a second, which would have been very slow if I had kept the original method of manually copying the bytes from the whole banks across.

In terms of the actual “swapping” process of active cartridge data, one way that cartridge memory mappers know when to swap active program or character banks is by listening for writes to particular addresses (like writes to areas of CPU or PPU memory which are mapped to cartridge ROM), which should never happen in “normal” execution. The cartridge memory mappers can treat the nonsense write request as really a request to switch the currently active program or character banks, and not as a real write request.

By using callbacks when memory access attempts are made, it makes it easy to model memory mappers by setting up callback write listeners that can change the currently active bank index when write requests are made to ROM, and also setting up callback read listeners for areas of CPU or PPU memory which map to cartridge ROM, so that they can return data bytes from cartridge ROM based on whatever the previously setup active bank variable was, without having to copy the entire bank data across.

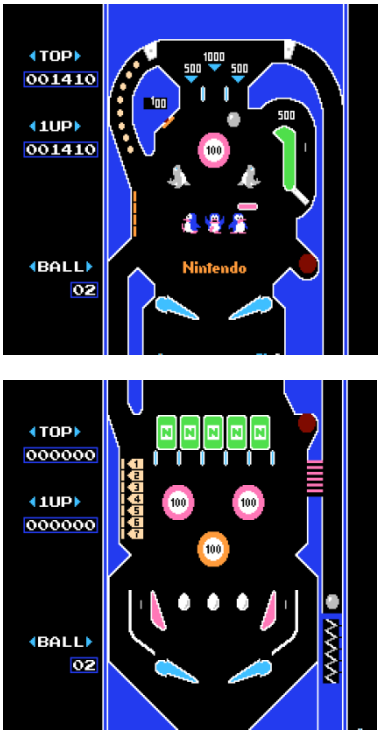
As NES game development progressed, the games became more complicated and in order to store more program code, more image code or produce more interesting effects (like more colours on screen at once), hundreds of different memory mappers were created. I've only written one of them, the first one Nintendo created, which was widespread and lets many early games run. I designed the emulation system with memory mappers in mind from the start and as such it has hooks which should allow arbitrary mappers to be created easily by just setting up specific callbacks. New memory mappers should fit in fairly seamlessly with the current design without having to modify much of the existing code, one would just need to write a new initialiser for the mapper and add a function reference to it in the existing mapper callback array.

Emulation of the PPU (Picture Processing Unit)

Games like Bomberman, Pinball, Tennis, Pacman, Balloon Fight and the original Mario Bros. game currently look and play fine, as they flip between different background screens (nametables), and don't partially scroll them in vertically or horizontally to overlap them.

For example, in Pinball a game board is represented as two screens, but you can only see one screen at a time, there is no scrolling between them. By scrolling I mean, it is never the case that the PPU is asked to render a frame which was formed by aligning parts of two separate nametables together, eg. the first 100 pixels of one nametable with the last 156 pixels of another nametable to form 256 horizontal pixels of output. To describe this process we would say that one nametable was "scrolling" to the left, out of the visible screen, and another nametable was "scrolling" into the visible screen from the right.

Instead, in Pinball if the ball goes high enough on the bottom screen, the PPU flips to the top screen, and if the ball goes low enough on the top screen, the PPU flips back to the bottom screen. There is no overlap or scrolling between them.



On the other hand, games which attempt to scroll background data in, currently appear to "jump" between screens when you cross the screen (nametable) boundary. This makes them largely unplayable as elements on the new screen may take you by surprise. For example, if a pit of death was scrolling towards you in Super Mario Bros., depending on when the screen flip occurred you might not see it until you were right above it. This adds a new, exciting challenge to games.

(Side note: in Super Mario Bros. you can still see enemies slowly plodding towards you from afar, as they are sprites, not background data, and the sprite rendering is based on arbitrary screen coordinates for up to 64 sprites and not on concepts of entire screen worth of data (nametables) which are being scrolled left/right or up/down.)

Lots of scrolling tricks (and other raster tricks, like split screen rendering or wavy screen distortions) are accomplished by the game changing (corrupting) the memory data of the PPU during the frame rendering (instead of between frame rendering), and hence these effects rely on precise timing of the PPU tile prefetch cycles in order for the CPU to know when to change the video data. To get scrolling to work properly I would need to change the way screens were rendered, to be more similar to the original NES PPU prefetch cycle where data for the next scanline was fetched on the current scanline and shuffled between an assortment of temporary buffers to eventually be rendered at a parametric later date.

I could try to make hacky tweaks to my current renderer to simulate those cycles -- I did attempt to do that briefly and created a fantastic mess for myself -- but realistically I should probably rewrite the rendering code again from scratch and design it to be more faithful from the start. The current design was more focused on working out clean and reusable ways to translate the background and sprite data from a collection of bits in memory into colours for the pixels on the screen, with less emphasis on emulating the exact process and order of operations the original NES PPU was using to do that. The actual NES PPU keeps an ongoing tile address to access which is modified in all kinds of bizarre ways by timed writes to PPU status registers, where the 16

bits in the tile address are actually formed by an assortment of 3 bits from one register, 5 bits from another register, 8 bits from another register but only at the start of a frame, etc. It is a horrible mess and to emulate it properly I would really need to start the renderer again with full knowledge of the mess in impending mind from the beginning, instead of having it take me by surprise like it currently has.

The games that appear and play correctly at the moment are by and large ones which respect the VBLANK (vertical blank) signal at the end of the frame as the only time they should modify video memory. For 20 scanlines after VBLANK starts, the PPU promises not to touch its memory at all, which gives the CPU approx. 1700 CPU cycles in order to modify the video memory. On VBLANK an interrupt is generated by the PPU and the CPU can catch this interrupt and trigger an interrupt handler whose purpose is to update video memory. Well, I say "the CPU can catch the interrupt", but the VBLANK interrupt actually triggers a NMI (non-maskable interrupt) which cannot be ignored, so if an NMI is generated the CPU **must** catch it. The one respite the CPU has is by setting a flag in the PPU registers which tells the PPU not to trigger NMIs during VBLANK in the first place.

In any case, if games only modify video data in that VBLANK period after catching an NMI, then it doesn't matter so much in what order the PPU fetches and renders data during a frame since the game will leave the PPU alone while it is doing the actual rendering. This allowed me to write the initial renderer in a relatively clean and simple way. However, I still attempted to allow the CPU to change and read video data during a frame update and not just during VBLANK - for example, when rendering sprites it would be most efficient to iterate over the 64 sprites once per frame and plot them out on the 2D screen buffer, but instead I iterate over them once at the start of each scanline and plot them out just for that 1D scanline. This allows the CPU to change their attributes between each scanlines, but it makes the sprite rendering roughly 240 times slower than it would be otherwise, as there are 240 visible scanlines.

Joypads

The two joypads are mapped to CPU memory space 0x4016 and 0x4017, respectively. The CPU will first write to 0x4016 to start a joystick "strobe", which asks the joypad to record its current state for the 4 directional buttons and 4 action buttons. Then the CPU can read from addresses 0x4016 or 0x4017 to read the recorded state of particular buttons, in a specific order (A, B, Select, Start, Up, Down, Left, Right).

It seems weird to me that, in order to do this, the CPU must perform 8 read operations, each returning a byte. If the byte has the 0^{th} bit set to 1, the button state is active, otherwise it is not. The rest of the byte data is ignored. For a system so obsessed with efficiency and compressing information (like the gloriously complicated attribute tables with their sub-tables), it isn't clear to my why the NES doesn't just do 1 byte read instead, with the 8 bits of the byte representing the state of the 8 different buttons. I guess it makes the CPU operations a bit simpler, since they only have to check whether each number is 0 by checking the zero status flag, instead of pulling apart and checking the individual bits using shift operations. It seems like an odd tradeoff.

GUI

The GUI periodically checks for button presses to set their state (by periodically I mean the NES periodically asks the GUI to do this, once every instruction). The joypads can query the recorded GUI state when asked by the CPU what their controller state is. The GUI is double buffered, and allows pixels to be drawn in arbitrary positions to a temporary buffer, which can then be switched with the active (visible) buffer by calling *gui_refresh*. This allows the entire screens worth of pixels to be calculated locally before attempting to communicate with the video card. I did originally try refreshing the screen after each individual pixel write, and it was fantastically slow.

Frame captures from old versions:

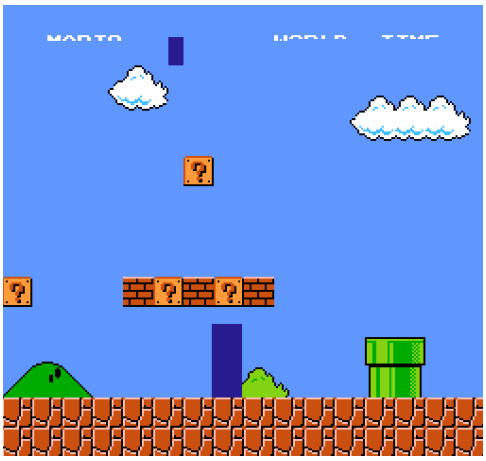
Taken when the nametable had just started to work, but before the pattern tables worked

Taken before colours worked

Taken when only 2 bits of colour worked

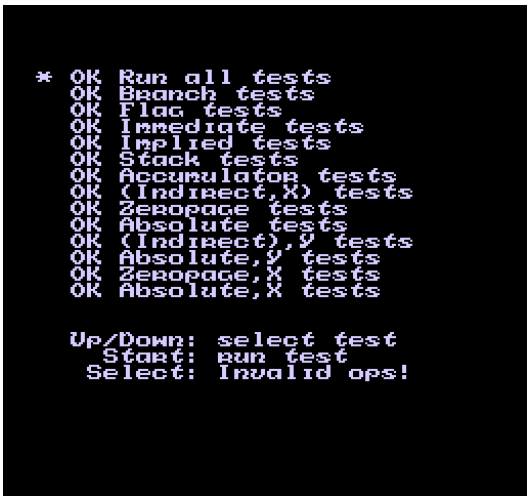
Another shot from when only 2 bits of colour worked

Taken before sprite rendering worked

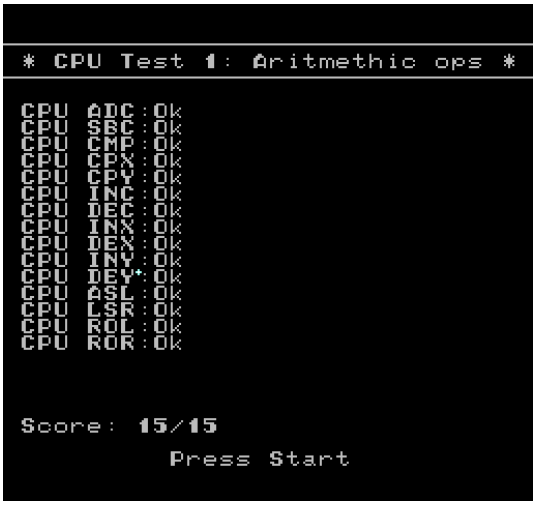


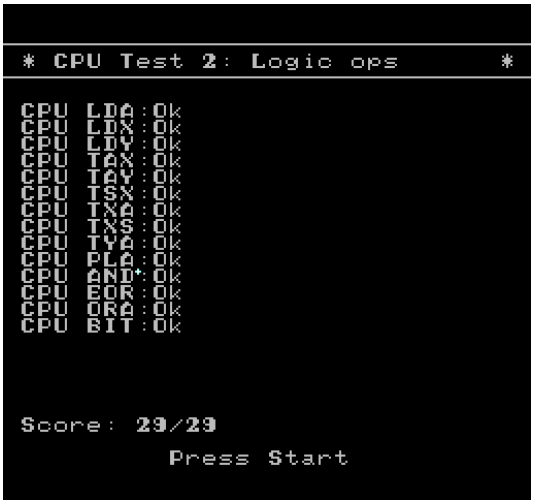
Taken when only sprite outlines were working (the purple rectangle is Mario's sprite)

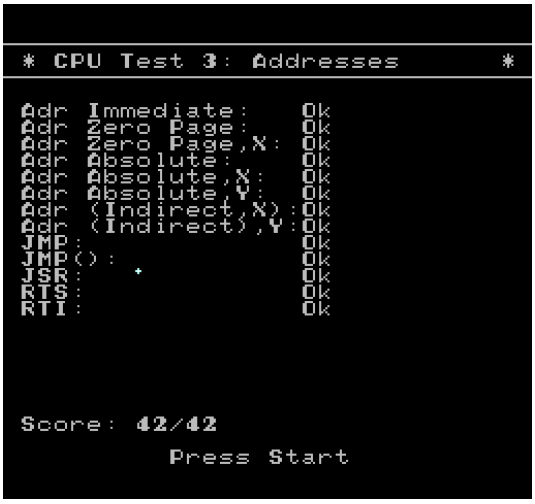
Frame captures from current version:

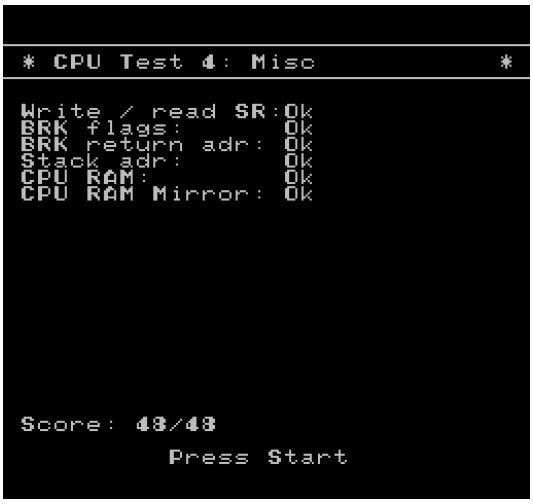


Passing all the CPU tests in the nestest.nes test ROM



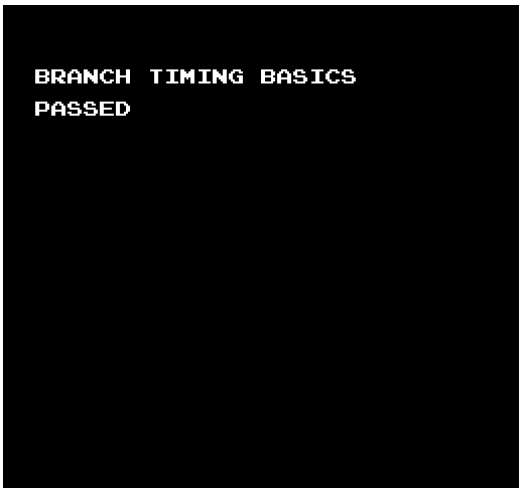






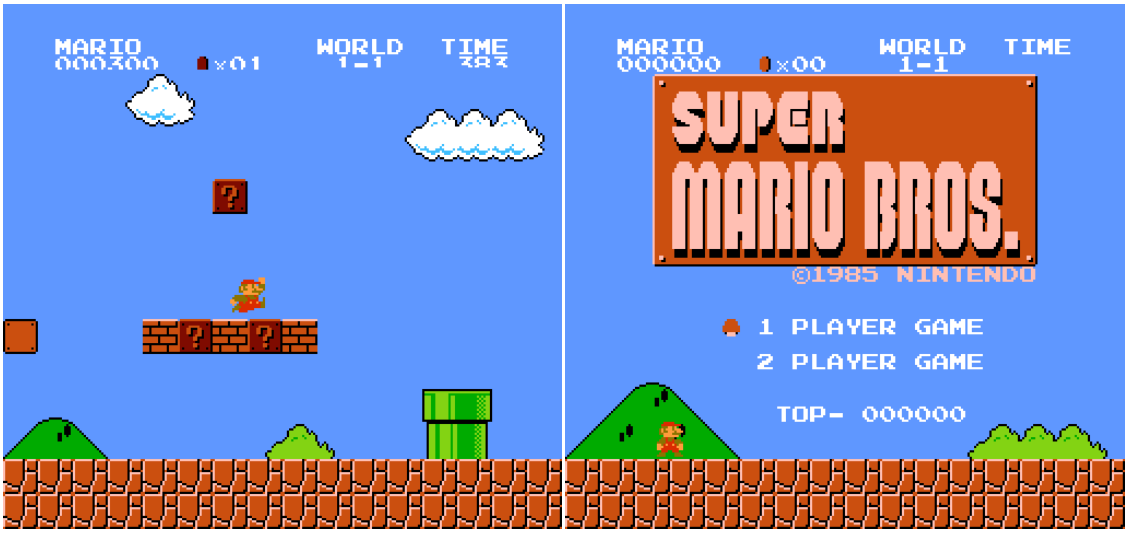
Passing all the CPU tests in the nesstress.nes test ROM

Passing some of the PPU tests in the nesstress.nes test ROM. The PPU is definitely not perfect!

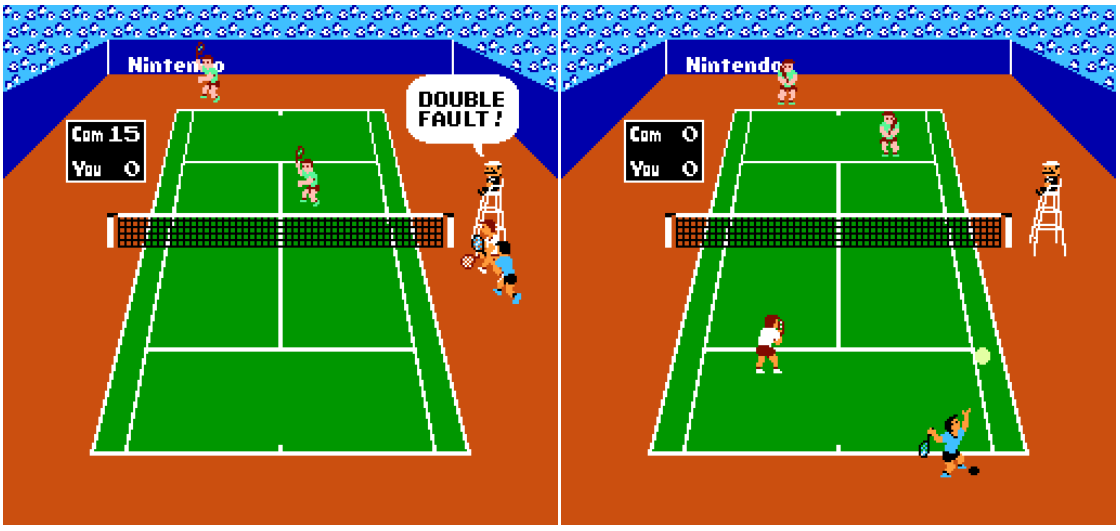


Passing a simple CPU cycle branch timing test ROM

Soccer title screen in the current version

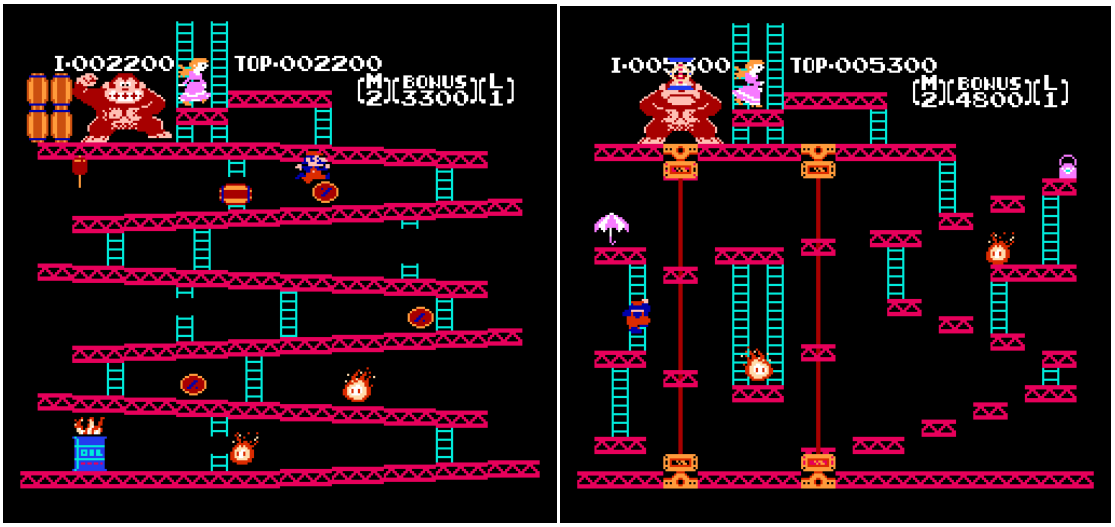


How Super Mario Bros. looks in the current version. Note that the static screens do not demonstrate the broken horizontal scrolling.



Tennis in the current version. The game emulates “perfectly.” On the left screenshot I am trying to beat the referee to death with a tennis racket, something my brother and I would often do when playing the game as children.

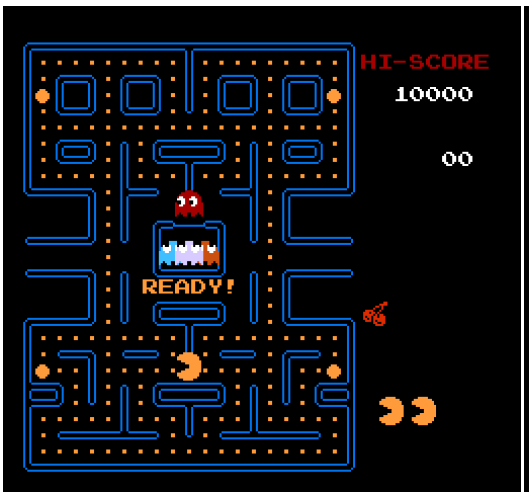
Excitebike. Note that, like Super Mario Bros., the horizontal scrolling has annoying distortions, even though it looks fine in this shot.

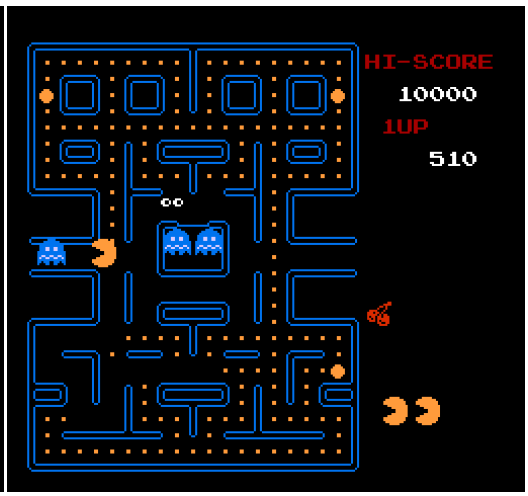


Scenes from Donkey Kong, the Nintendo game that started it all. The game emulates “perfectly.”

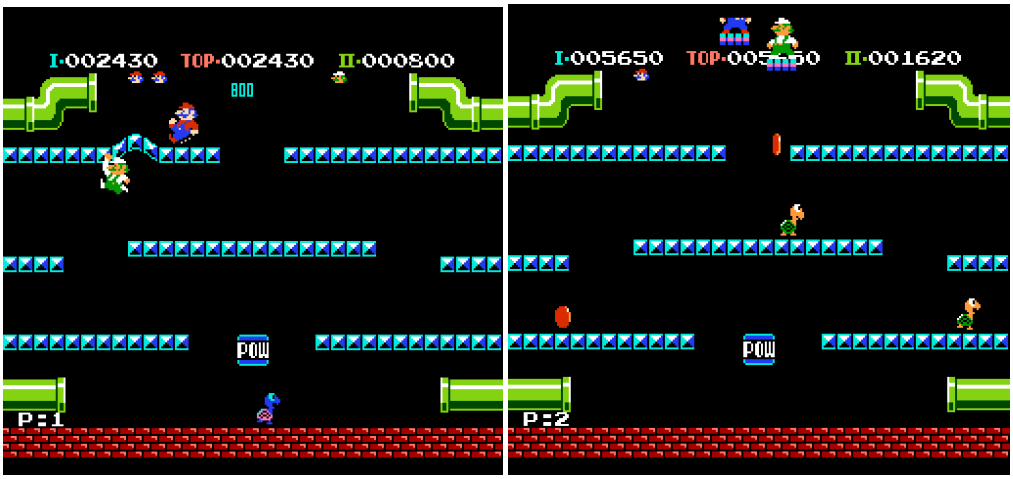
Scenes from Duck Hunt. It uses a light gun controller which I have not emulated, so the duck always escapes!

Scenes from Urban Champion, which is largely playable. Punches won't connect sometimes, but I seem to remember that happening in the original game as well. I really like the title screen.

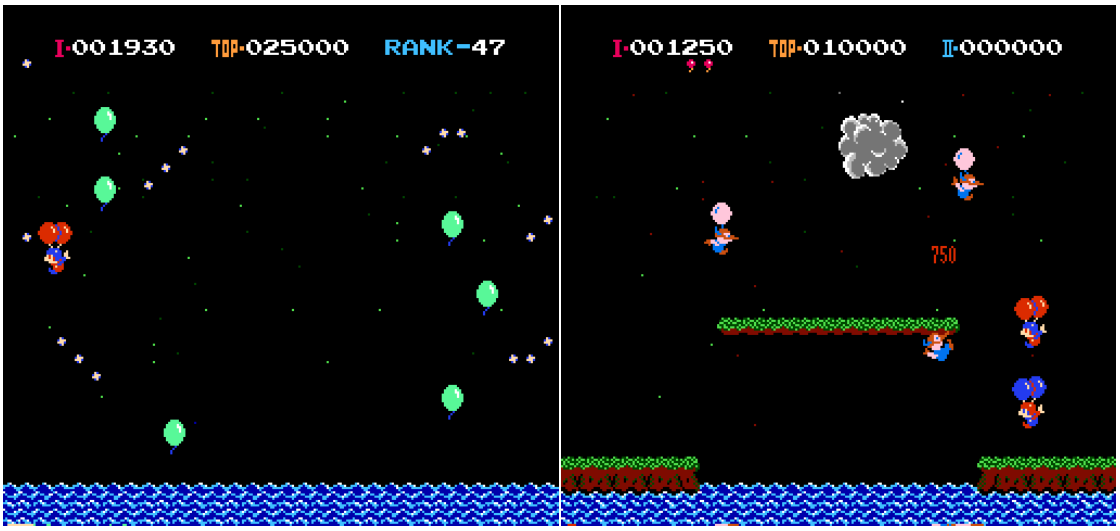




Hunting down ghosts in Pacman. The game emulates "perfectly."



Playing the original Mario Bros. game with my Dad after setting up the 2nd player emulation.



Playing Balloon Fight. The first screen illustrates sprite scrolling, which works fine (it is just nametable scrolling that has issues.) The second screen is from 2-player mode, and I am about to pop my Dad's balloons.

Notorious problems encountered

The main problem was the lack of reliable information on the hardware. Explanations were given that were often misguided, and sometimes contradictory. For example on the same page of a very official looking 6502 operation data sheet (see appendix), an Indirect Jump is given two different OP codes, 0xFF and 0x6C... at the moment I allow both to match, but it seems strange that one operation would have two different opcodes. On another specification, I was told that the 5th bit of the 6502 CPU status register was always ignored, you could never set it or read it... yet another document said that it should always have the value 1, offering no reason why.

Another document on querying the joypads suggested that the first 8 reads to a joypad should return the button state for those 8 buttons, and every subsequent read should return 1. But another document says that, in any group of 20 reads from a joypad, the first 8 reads are for joypad n, the next 8 reads are for joypad n+2 (so joypads 1 and 3 would share the same socket), and the following 4 reads were for connectivity information. Theoretically, if I did always return 1 for every read after the 8th read, that would probably be OK, since nearly no games use the 3rd and 4th joypads as you need a special adapter to plug them in. So, in most cases the “pro tip” to always return 1 would have been fine. But overall it was still incorrect, and I'm scared that there are other things like that that have made it into the code based on specifications I have read that were actually wrong.

My strategy so far has been to find as many different documents about the same hardware as possible and try to implement the common parts, but this still doesn't always work out. Different documents will discuss different pitfalls that the authors have discovered, which aren't necessarily confirmed in others. For instance, that if a status register is pushed on to the stack, it should have the "breakpoint flag" set to 1. But others will argue that there is actually no such thing as a "breakpoint flag", since there are no operations to set or clear it like other CPU status flags.

And, although Nintendo adopted the MOS 6502 chip as the core design for the CPU, they did make subtle changes that make it behave differently to the original chip which are hard to find information on. For example certain addressing modes cause lower bits to wrap around locally without affecting the higher bits, like an Indirect Jump, but that does not happen on original MOS 6502 hardware.

As much as I have just complained about inconsistent CPU documentation.. the CPU has much better documentation available than the picture processing unit, which in turn has much better documentation available than the audio processing unit. All of this is frustrating, but at the same time it is utterly wonderful that information is available on these things at all; otherwise I would have had to take apart my real NES hardware, and also get an electrical engineering degree in order to understand what I was doing. So I am still very grateful to the random assortment of documentation available on the internet for letting me avoid that process, however inconsistent it is.

Future directions

- Create a more accurate Picture Processing Unit. Priority #1 would be to get nametable scrolling working correctly. To emulate the PPU 100% accurately requires a lot of overhead; I've read about emulators which do this for the NES - an 8 bit console - as running more slowly than emulators written for the following 16bit generation of consoles. It would still be worthwhile doing.

- Attempt to emulate the Audio Processing Unit. Many technical documents I read skipped over this, with sections like "Audio: to be written", so I gave it a low priority. However there is still some information freely available on it and it would be fun to try to generate 8 bit music.

- Create more memory mappers for cartridge banks. I've written a mapper for NROM (the original mapper), which is fine for a lot of early games. However many later games won't load because they used exotic memory mappers, even though the games themselves might work fine once loaded. The effort here to write more mappers should be minimal given the approach I have taken with the design of letting each mapper setup arbitrary memory callbacks. As yet I have not attempted to write other mappers because I wanted to get the games which used the existing NROM mapper working perfectly first before creating new problems for myself.

Conclusions

Writing this was a lot of fun and I'm sad that it is over. One of the coolest things for me was finding out exactly how the "Zapper" works, a light gun controller that I remember firing at my television screen as a 6 year old. (It works by the NES polling the gun controller to see if the trigger is pressed. If the trigger is pressed, the NES very briefly renders white boxes where objects which could be shot at are positioned, and black everywhere else, and then asks the gun whether it is currently looking at white pixels. This happens so fast that I never saw the white boxes as a kid, but I will have to plug my original NES in again and watch for them now).

I'm very happy with what I achieved in this project. I have tried to evenly balance design and style issues against accuracy and timing issues, especially for the CPU, but overall with more emphasis on design than on efficiency. There are plenty of sections of the code which would be faster if I violated abstraction, however they generally involve small constant overheads (like having to request access to an ADT through a wrapper rather than speaking directly to it). I've generally tried to keep the overall complexity of operations low while accepting small constant overheads like that if they made the design cleaner.

There are plenty of things which could be improved, for sure. But at the end of the day, I've still made an emulator which loads, renders and lets me play simple games from my childhood, which is just amazing to me. It was also a lot of fun to play around with function pointers, macros, bitwise operators and all kinds of C trickery in order to get this working. Some people seem more shocked that I wrote the emulator in C, rather than shocked about the fact that I wrote an emulator at all. C is great.

References & Thanks

- The most useful and reliable source of CPU information was a personal website someone named Andrew Jacobs made about his favourite processor, the NMOS 6502

  http://www.obelisk.demon.co.uk/6502/

- This list of opcodes, operands and cycles was also very useful, and was the initial basis of the CPU core. So much information on one page…

  http://ericclever.com/6500/

- There wasn't really a single source of really reliable PPU information, but lots of documents on this NES emulation page covered different aspects:

  http://nesdev.parodius.com/

- They also have a related wiki and forums:

  http://nesdevwiki.org/wiki/index.php/NESdevWiki

  http://nesdev.parodius.com/bbs/

- Possibly the most useful source of information on the whole *nesdev* site was this document:

  http://nesdev.parodius.com/NESDoc.pdf

  It skips some implementation details and gets a fact facts wrong but is gives a very good general overview of different aspects of the system. (Note that, like many other documents, it does not cover sound emulation.

- Aleksandar Ignjatovic for approving this project as I had so much fun developing it.

Appendix:

List of NROM games that should load with the NROM memory mapper:

10 Yard Fight	Nintendo	NROM
1942	Capcom	NROM
Balloon Fight	Nintendo	NROM
Baseball	Nintendo	NROM
Bomberman	Hudson	NROM
Burger Time	Data East	NROM
Chubby Cherub	Bandai	NROM
Defender 2	HAL	NROM
Dig Dug 2	Bandai	NROM
Donkey Kong	Nintendo	NROM
Donkey Kong 3	Nintendo	NROM
Donkey Kong Jr	Nintendo	NROM
Donkey Kong Jr Math	Nintendo	NROM
Duck Hunt	Nintendo	NROM
Elevator Action	Taito	NROM
Excitebike	Nintendo	NROM
Galaga	Bandai	NROM
Golf	Nintendo	NROM
Gyromite	Nintendo	NROM
Hogan's Alley	Nintendo	NROM
Hydlide	FCI	NROM
Ice Climber	Nintendo	NROM
Ice Hockey	Nintendo	NROM
Kung Fu	Nintendo	NROM
Lode Runner	Broderbund	NROM
Lunar Pool	FCI	NROM
Mach Rider	Nintendo	NROM
Magmax	FCI	NROM
Mario Brothers	Nintendo	NROM
Millipede	HAL	NROM
Ms. Pacman	Namco	NROM
Muscle	Bandai	NROM
Othello	Acclaim	NROM
Pac Man (licensed)	Tengen	NROM
Pinball	Nintendo	NROM
Popeye	Nintendo	NROM
Raid on Bungling Bay	Broderbund	NROM
Seicross	FCI	NROM
Slalom	Nintendo	NROM
Soccer	Nintendo	NROM
Spelunker	Broderbund	NROM
Spy vs. Spy	Kemco	NROM
Sqoon	Irem	NROM
Super Mario Brothers	Nintendo	NROM
Tag Team Wrestling	Data East	NROM
Tennis	Nintendo	NROM
Urban Champion	Nintendo	NROM
Volleyball	Nintendo	NROM
Wild Gunman	Nintendo	NROM
Wrecking Crew	Nintendo	NROM
Xevious	Bandai	NROM

apu.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "globals.h"
#include "apu.h"
#include "nes.h"

struct apu {
  Byte pulse1_control;
  Byte pulse1_rampControl;
  Byte pulse1_fineTune;
  Byte pulse1_courseTune;

  Byte pulse2_control;
  Byte pulse2_rampControl;
  Byte pulse2_fineTune;
  Byte pulse2_courseTune;

  Byte triangle_control1;
  Byte triangle_control2;

  Byte triangle_frequency1;
  Byte triangle_frequency2;

  Byte noise_control1;

  Byte noise_frequency1;
  Byte noise_frequency2;

  Byte delta_control;
  Byte delta_da;
  Byte delta_address;
  Byte delta_dataLength;
};

APU apu_init(void) {
  APU apu = (APU) malloc(sizeof(struct apu));
  assert(apu != NULL);

  apu->pulse1_control = 0;
  apu->pulse1_rampControl = 0;
  apu->pulse1_fineTune = 0;
  apu->pulse1_courseTune = 0;

  apu->pulse2_control = 0;
  apu->pulse2_rampControl = 0;
  apu->pulse2_fineTune = 0;
  apu->pulse2_courseTune = 0;

  apu->triangle_control1 = 0;
  apu->triangle_control2 = 0;

  apu->triangle_frequency1 = 0;
  apu->triangle_frequency2 = 0;

  apu->noise_control1 = 0;

  apu->noise_frequency1 = 0;
  apu->noise_frequency2 = 0;

  apu->delta_control = 0;
  apu->delta_da = 0;
  apu->delta_address = 0;
  apu->delta_dataLength = 0;

  return apu;
}

void apu_destroy(APU apu) {
```

```
  assert(apu != NULL);
  free(apu);
}

void apu_step(NES nes) {
  assert(nes != NULL);

  // not implemented
  // debug_printf("apu_step\n");
}
```

apu.h

```
#include "apu_type.h"

#include "nes_type.h"

APU apu_init(void);

void apu_step(NES nes);

void apu_destroy(APU apu);
```

apu_type.h

```
#ifndef APU_TYPE_H
  #define APU_TYPE_H
  typedef struct apu *APU;
#endif
```

cartridge.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <stdint.h>
#include "globals.h"
#include "memory.h"
#include "cartridge.h"

struct characterBank {
  Byte bytes[BYTES_PER_CHARACTER_BANK];
};

typedef struct characterBank *CharacterBank;

struct programBank {
  Byte bytes[BYTES_PER_PROGRAM_BANK];
};

typedef struct programBank *ProgramBank;

struct cartridge {
  Byte numProgramBanks;
  Byte numCharacterBanks;
  MirrorType mirrorType;
  Boolean hasBatteryPack;
  Boolean hasTrainer;
  Byte mmuNumber;
  Byte numRamBanks;
  CharacterBank *characterBanks;
  ProgramBank *programBanks;
  Byte title[MAX_TITLE_BYTES];
};

void cartridge_print(Cartridge cartridge) {
  assert(cartridge != NULL);
  debug_printf("\n");
  debug_printf("Cartridge details:\n");
  debug_printf("  numProgramBanks: %d\n", cartridge->numProgramBanks);
  debug_printf("  numCharacterBanks: %d\n", cartridge->numCharacterBanks);
  debug_printf("  mirrorType: %d\n", cartridge->mirrorType);
  debug_printf("  hasBatteryPack: %d\n", cartridge->hasBatteryPack);
  debug_printf("  hasTrainer: %d\n", cartridge->hasTrainer);
  debug_printf("  mmuNumber: %d\n", cartridge->mmuNumber);
  debug_printf("  numRamkBanks: %d\n", cartridge->numRamBanks);
  debug_printf("  title: %s\n", cartridge->title);
}

typedef struct header *Header;

struct header {
  Byte magicLabel[3];
  Byte magicValue;
  Byte numProgramBanks;
  Byte numCharacterBanks;
  Byte controlByte1;
  Byte controlByte2;
  Byte numRamBanks;
  Byte magicReservedValues[7];
};

Byte cartridge_getNumProgramBanks(Cartridge cartridge) {
  assert(cartridge != NULL);
  return cartridge->numProgramBanks;
}

Byte cartridge_readProgramBank(Cartridge cartridge, Byte bank, Address address) {
  assert(cartridge != NULL);
  assert(bank < cartridge->numProgramBanks);
  assert(address < BYTES_PER_PROGRAM_BANK);
```

```
  return cartridge->programBanks[bank]->bytes[address];
}

Byte cartridge_readCharacterBank(Cartridge cartridge, Byte bank, Address address) {
  assert(cartridge != NULL);
  assert(bank < cartridge->numCharacterBanks);
  assert(address < BYTES_PER_CHARACTER_BANK);

  return cartridge->characterBanks[bank]->bytes[address];
}

static void  parseHeader(Cartridge cartridge, FILE *file) {
  assert(cartridge != NULL);
  assert(file != NULL);
  fseek(file, 0, SEEK_SET);

  Header header = malloc(sizeof(struct header));

  assert(fread(header, sizeof(struct header), 1, file) == 1);

  assert(header->magicLabel[0] == 'N');
  assert(header->magicLabel[1] == 'E');
  assert(header->magicLabel[2] == 'S');
  assert(header->magicValue == 0x1A);

  cartridge->numProgramBanks = header->numProgramBanks;
  assert(cartridge->numProgramBanks > 0);

  cartridge->numCharacterBanks = header->numCharacterBanks;
  cartridge->numRamBanks = header->numRamBanks;

  if ((header->controlByte1 & MASK_BIT3) == MASK_BIT3) {
    cartridge->mirrorType = BOTH;
  } else if ((header->controlByte1 & MASK_BIT0) == 0) {
    cartridge->mirrorType = HORIZONTAL;
  } else if ((header->controlByte1 & MASK_BIT0) == MASK_BIT0) {
    cartridge->mirrorType = VERTICAL;
  } else {
    assert(FALSE);
  }

  if ((header->controlByte1 & MASK_BIT1) == MASK_BIT1) {
    cartridge->hasBatteryPack = 1;
  } else {
    cartridge->hasBatteryPack = 0;
  }

  if ((header->controlByte1 & MASK_BIT2) == MASK_BIT2) {
    cartridge->hasTrainer = 1;
  } else {
    cartridge->hasTrainer = 0;
  }

  cartridge->mmuNumber = (header->controlByte1 & (MASK_BIT4 | MASK_BIT5 | MASK_BIT6 |
MASK_BIT7)) >> 4;
  cartridge->mmuNumber += header->controlByte2 & (MASK_BIT4 | MASK_BIT5 | MASK_BIT6 |
MASK_BIT7);

  assert((header->controlByte2 & MASK_BIT0) == 0);
  assert((header->controlByte2 & MASK_BIT1) == 0);
  assert((header->controlByte2 & MASK_BIT2) == 0);
  assert((header->controlByte2 & MASK_BIT3) == 0);

  int i;
  for (i=0; i < 7; i++) {
    assert(header->magicReservedValues[i] == 0);
  }

  fseek(file, sizeof(struct header), SEEK_SET);
  if (cartridge->hasTrainer) {
    fseek(file, TRAINER_BYTES, SEEK_CUR);
```

```
  }

  free(header);
}

static void parseProgramBanks(Cartridge cartridge, FILE *file) {
  assert(cartridge != NULL);
  assert(file != NULL);

  cartridge->programBanks = (ProgramBank*) malloc(sizeof(ProgramBank) * cartridge-
>numProgramBanks);
  assert(cartridge->programBanks != NULL);

  int i;
  for (i=0; i < cartridge->numProgramBanks; i++) {
    ProgramBank programBank = malloc(sizeof(struct programBank));
    assert(programBank != NULL);
    assert(fread(programBank, sizeof(struct programBank), 1, file) == 1);
    cartridge->programBanks[i] = programBank;
  }

}

static void parseCharacterBanks(Cartridge cartridge, FILE *file) {
  assert(cartridge != NULL);
  assert(file != NULL);

  cartridge->characterBanks = (CharacterBank*) malloc(sizeof(CharacterBank) * cartridge-
>numCharacterBanks);
  assert(cartridge->characterBanks != NULL);

  int i;
  for (i=0; i < cartridge->numCharacterBanks; i++) {
    CharacterBank characterBank = malloc(sizeof(struct characterBank));
    assert(characterBank != NULL);
    assert(fread(characterBank, sizeof(struct characterBank), 1, file) == 1);
    cartridge->characterBanks[i] = characterBank;
  }
}

static void parseTitle(Cartridge cartridge, FILE *file) {
  assert(cartridge != NULL);
  assert(file != NULL);

  fread(cartridge->title, sizeof(Byte), 128, file);
}


Cartridge cartridge_init(char *filename) {
  Cartridge cartridge = malloc(sizeof(struct cartridge));
  assert(cartridge != NULL);

  FILE *file = fopen(filename, "rb");
  assert(file != NULL);

  cartridge->numProgramBanks = 0;
  cartridge->numCharacterBanks = 0;
  cartridge->mirrorType = HORIZONTAL;
  cartridge->hasBatteryPack = FALSE;
  cartridge->hasTrainer = FALSE;
  cartridge->mmuNumber = 0;
  cartridge->numRamBanks = 0;
  cartridge->characterBanks = NULL;
  cartridge->programBanks = NULL;
  cartridge->title[0] = 0;

  parseHeader(cartridge, file);
  parseProgramBanks(cartridge, file);
  parseCharacterBanks(cartridge, file);

  parseTitle(cartridge, file);
```

```
  fclose(file);

  return cartridge;
}

void cartridge_destroy(Cartridge cartridge) {
  assert(cartridge != NULL);

  int i;
  for (i=0; i < cartridge->numCharacterBanks; i++) {
    free(cartridge->characterBanks[i]);
  }

  for (i=0; i < cartridge->numProgramBanks; i++) {
    free(cartridge->programBanks[i]);
  }

  free(cartridge);
}

Byte cartridge_getMMUNumber(Cartridge cartridge) {
  assert(cartridge != NULL);
  return cartridge->mmuNumber;
}

Byte cartridge_getNumCharacterBanks(Cartridge cartridge) {
  assert(cartridge != NULL);
  return cartridge->numCharacterBanks;
}

MirrorType cartridge_getMirrorType(Cartridge cartridge) {
  assert(cartridge != NULL);
  return cartridge->mirrorType;
}
```

cartridge.h

```
#include "cartridge_type.h"
#include "globals.h"

#define BYTES_PER_CHARACTER_BANK (BYTES_PER_KILOBYTE * 8)

#define BYTES_PER_PROGRAM_BANK (BYTES_PER_KILOBYTE * 16)

#define MAX_TITLE_BYTES 128
#define TRAINER_BYTES 512

// affects how ppu memory address spaces are mirrored
typedef enum {
  HORIZONTAL = 0,
  VERTICAL = 1,
  BOTH = 2
} MirrorType;

Cartridge cartridge_init(char *filename);

void cartridge_print(Cartridge cartridge);

void cartridge_destroy(Cartridge cartridge);

Byte cartridge_getMMUNumber(Cartridge cartridge);

Byte cartridge_getNumProgramBanks(Cartridge cartridge);
Byte cartridge_getNumCharacterBanks(Cartridge cartridge);

MirrorType cartridge_getMirrorType(Cartridge cartridge);

Byte cartridge_readProgramBank(Cartridge cartridge, Byte bank, Address address);
Byte cartridge_readCharacterBank(Cartridge cartridge, Byte bank, Address address);
```

cartridge_type.h

```
#ifndef CARTRIDGE_TYPE_H
  #define CARTRIDGE_TYPE_H
  typedef struct cartridge *Cartridge;
#endif
```

colour.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "globals.h"
#include "colour.h"

Colour colour_init(Byte red, Byte green, Byte blue) {
  Colour colour = (Colour) malloc(sizeof(struct colour));
  assert(colour != NULL);

  colour_setColour(colour, red, green, blue);

  return colour;
}

void colour_destroy(Colour colour) {
  assert(colour != NULL);
  free(colour);
}

void colour_setRed(Colour colour, Byte red) {
  assert(colour != NULL);
  colour->red = red;
}

void colour_setGreen(Colour colour, Byte green) {
  assert(colour != NULL);
  colour->green = green;
}

void colour_setBlue(Colour colour, Byte blue) {
  assert(colour != NULL);
  colour->blue = blue;
}

Byte colour_getRed(Colour colour) {
  assert(colour != NULL);
  return colour->red;
}

Byte colour_getGreen(Colour colour) {
  assert(colour != NULL);
  return colour->green;
}

Byte colour_getBlue(Colour colour) {
  assert(colour != NULL);
  return colour->blue;
}

void colour_setColour(Colour colour, Byte red, Byte green, Byte blue) {
  assert(colour != NULL);

  colour->red = red;
  colour->green = green;
  colour->blue = blue;
}
```

colour.h

```
#include "colour_type.h"

#include "globals.h"

void colour_destroy(Colour colour);

void colour_setRed(Colour colour, Byte red);

void colour_setGreen(Colour colour, Byte green);

void colour_setBlue(Colour colour, Byte blue);

Byte colour_getRed(Colour colour);

Byte colour_getGreen(Colour colour);

Byte colour_getBlue(Colour colour);

Colour colour_init(Byte red, Byte green, Byte blue);

void colour_setColour(Colour colour, Byte red, Byte green, Byte blue);
```

colour_type.h

```
#ifndef COLOUR_TYPE_H

  #define COLOUR_TYPE_H

  typedef struct colour *Colour;

  // we break abstraction here so that ppu.c can declare an array of 'struct colour' for
the global system palette
  // maybe find a better way to do this later

  struct colour {
    Byte red;
    Byte green;
    Byte blue;
  };

#endif
```

cpu.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "globals.h"
#include "cpu.h"
#include "nes.h"
#include "instructions.h"
#include "cpuMemory.h"

// many comments below are based on operational details from
http://www.obelisk.demon.co.uk/6502/

struct cpu {
  Address programCounter;
  Byte stackPointer;
  Byte status;
  Byte indexX;
  Byte indexY;
  Byte accumulator;
};

CPU cpu_init(void) {
  CPU cpu = (CPU) malloc(sizeof(struct cpu));
  assert(cpu != NULL);

  cpu->programCounter = 0;
  cpu->status = CPU_STATUS_REGISTER_INITIAL_VALUE;
  cpu->indexX = 0;
  cpu->indexY = 0;
  cpu->accumulator = 0;
  cpu->stackPointer = CPU_STACK_POINTER_INITIAL_VALUE;

  return cpu;
}

void cpu_destroy(CPU cpu) {
  assert(cpu != NULL);
  free(cpu);
}

Address cpu_getProgramCounter(CPU cpu) {
  assert(cpu != NULL);
  return cpu->programCounter;
}

Byte cpu_getStackPointer(CPU cpu) {
  assert(cpu != NULL);
  return cpu->stackPointer;
}

Byte cpu_getStatus(CPU cpu) {
  assert(cpu != NULL);

  // 1 at all times
  cpu->status |= MASK_BIT5;

  return cpu->status;
}

Byte cpu_getIndexX(CPU cpu) {
  assert(cpu != NULL);
  return cpu->indexX;
}

Byte cpu_getIndexY(CPU cpu) {
  assert(cpu != NULL);
  return cpu->indexY;
}
```

```
Byte cpu_getAccumulator(CPU cpu) {
  assert(cpu != NULL);
  return cpu->accumulator;
}

void cpu_setProgramCounter(CPU cpu, Address programCounter) {
  assert(cpu != NULL);
  cpu->programCounter = programCounter;
}

void cpu_setStackPointer(CPU cpu, Byte stackPointer) {
  assert(cpu != NULL);
  cpu->stackPointer = stackPointer;
}

void cpu_setStatus(CPU cpu, Byte status) {
  assert(cpu != NULL);

  // 1 at all times
  status |= MASK_BIT5;

  cpu->status = status;
}

void cpu_setIndexX(CPU cpu, Byte indexX) {
  assert(cpu != NULL);
  cpu->indexX = indexX;
}

void cpu_setIndexY(CPU cpu, Byte indexY) {
  assert(cpu != NULL);
  cpu->indexY = indexY;
}

void cpu_setAccumulator(CPU cpu, Byte accumulator) {
  assert(cpu != NULL);
  cpu->accumulator = accumulator;
}

static void cpu_increaseProgramCounter(CPU cpu) {
  assert(cpu != NULL);
  cpu->programCounter++;
}

static void cpu_JMP(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("JMP\n");

  cpu->programCounter = address;

}

static Boolean cpu_getZero(CPU cpu) {
  assert(cpu != NULL);

  if ((cpu->status & MASK_STATUS_ZERO_ON) == MASK_STATUS_ZERO_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void cpu_setZero(CPU cpu, Boolean state) {
  assert(cpu != NULL);

  if (state == TRUE) {
    cpu->status |= MASK_STATUS_ZERO_ON;
  } else if (state == FALSE) {
```

```
    cpu->status &= MASK_STATUS_ZERO_OFF;
  }
}

//Z     Zero Flag      Set if A = 0
static void cpu_updateZero(CPU cpu, Byte data) {
  assert(cpu != NULL);

  if (data == 0) {
    cpu_setZero(cpu, TRUE);
  } else {
    cpu_setZero(cpu, FALSE);
  }
}

static void cpu_setDecimal(CPU cpu, Boolean state) {
  assert(cpu != NULL);

  if (state == TRUE) {
    cpu->status |= MASK_STATUS_DECIMAL_ON;
  } else if (state == FALSE) {
    cpu->status &= MASK_STATUS_DECIMAL_OFF;
  }
}

static Boolean cpu_getDecimal(CPU cpu) {
  assert(cpu != NULL);

  if ((cpu->status & MASK_STATUS_DECIMAL_ON) == MASK_STATUS_DECIMAL_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

// SED - Set Decimal Flag
static void cpu_SED(CPU cpu) {
  assert(cpu != NULL);

  cpu_setDecimal(cpu, TRUE);
}

// CLD - Clear Decimal Mode
static void cpu_CLD(CPU cpu) {
  assert(cpu != NULL);

  cpu_setDecimal(cpu, FALSE);
}

static Boolean cpu_getNegative(CPU cpu) {
  assert(cpu != NULL);

  if ((cpu->status & MASK_STATUS_NEGATIVE_ON) == MASK_STATUS_NEGATIVE_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void cpu_setNegative(CPU cpu, Boolean state) {
  assert(cpu != NULL);

  if (state == TRUE) {
    cpu->status |= MASK_STATUS_NEGATIVE_ON;
  } else if (state == FALSE) {
    cpu->status &= MASK_STATUS_NEGATIVE_OFF;
  }
}

//N     Negative Flag  Set if bit 7 of A is set
static void cpu_updateNegative(CPU cpu, Byte data) {
```

```
  assert(cpu != NULL);
  if ((data & MASK_BIT7) == MASK_BIT7) {
    cpu_setNegative(cpu, TRUE);
  } else {
    cpu_setNegative(cpu, FALSE);
  }
}

// LDA - Load Accumulator
// Loads a byte of memory into the accumulator setting the zero and negative flags as
appropriate.
static void cpu_LDA(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("LDA\n");

  Byte data = nes_readCPUMemory(nes, address);

  cpu_updateZero(cpu, data);

  cpu_updateNegative(cpu, data);

  cpu_setAccumulator(cpu, data);
}

// STA - Store Accumulator
// Stores the contents of the accumulator into memory.
static void cpu_STA(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("STA\n");

  nes_writeCPUMemory(nes, address, cpu->accumulator);
}

// LDX - Load X Register
// Loads a byte of memory into the X register setting the zero and negative flags as
appropriate.
static void cpu_LDX(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("LDX\n");

  Byte data = nes_readCPUMemory(nes, address);

  cpu_updateZero(cpu, data);

  cpu_updateNegative(cpu, data);

  cpu_setIndexX(cpu, data);
}

// STX - Store X Register
// Stores the contents of the X register into memory.
static void cpu_STX(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("STX\n");

  nes_writeCPUMemory(nes, address, cpu->indexX);

}

// LDY - Load Y Register
```

```
// Loads a byte of memory into the Y register setting the zero and negative flags as
appropriate.
static void  cpu_LDY(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("LDY\n");

  Byte data = nes_readCPUMemory(nes, address);

  cpu_updateZero(cpu, data);

  cpu_updateNegative(cpu, data);

  cpu_setIndexY(cpu, data);
}

// STY - Store Y Register
// Stores the contents of the Y register into memory.
static void cpu_STY(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("STY\n");

  nes_writeCPUMemory(nes, address, cpu->indexY);
}

// AND - Logical AND
// A logical AND is performed, bit by bit, on the accumulator contents using the
contents of a byte of memory.
static void cpu_AND(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("AND\n");

  Byte data = nes_readCPUMemory(nes, address);

  cpu->accumulator &= data;

  cpu_updateZero(cpu, cpu->accumulator);

  cpu_updateNegative(cpu, cpu->accumulator);
}

// ORA - Logical Inclusive OR
// An inclusive OR is performed, bit by bit, on the accumulator contents using the
contents of a byte of memory.
static void cpu_ORA(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("ORA\n");

  Byte data = nes_readCPUMemory(nes, address);

  cpu->accumulator |= data;

  cpu_updateZero(cpu, cpu->accumulator);

  cpu_updateNegative(cpu, cpu->accumulator);
}

// EOR - Exclusive OR
// An exclusive OR is performed, bit by bit, on the accumulator contents using the
contents of a byte of memory.
static void cpu_EOR(NES nes, Address address) {
```

```
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("EOR\n");

  Byte data = nes_readCPUMemory(nes, address);

  cpu->accumulator ^= data;

  cpu_updateZero(cpu, cpu->accumulator);

  cpu_updateNegative(cpu, cpu->accumulator);
}

static void cpu_setOverflow(CPU cpu, Boolean state) {
  assert(cpu != NULL);

  if (state == TRUE) {
    cpu->status |= MASK_STATUS_OVERFLOW_ON;
  } else if (state == FALSE) {
    cpu->status &= MASK_STATUS_OVERFLOW_OFF;
  }
}

static Boolean cpu_getOverflow(CPU cpu) {
  assert(cpu != NULL);

  if ((cpu->status & MASK_STATUS_OVERFLOW_ON) == MASK_STATUS_OVERFLOW_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

// BIT - Bit Test
// This instructions is used to test if one or more bits are set in a target memory
location.
// The mask pattern in A is ANDed with the value in memory to set or clear the zero flag,
but the result is not kept.
// Bits 7 and 6 of the value from memory are copied into the N and V flags.
static void cpu_BIT(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("BIT\n");

  Byte data = nes_readCPUMemory(nes, address);

  cpu_updateZero(cpu, cpu->accumulator & data);

  cpu_updateNegative(cpu, data);

  if ((data & MASK_BIT6) == MASK_BIT6) {
    cpu_setOverflow(cpu, TRUE);
  } else {
    cpu_setOverflow(cpu, FALSE);
  }
}

static void cpu_setCarry(CPU cpu, Boolean state) {
  assert(cpu != NULL);

  if (state == TRUE) {
    cpu->status |= MASK_STATUS_CARRY_ON;
  } else if (state == FALSE) {
    cpu->status &= MASK_STATUS_CARRY_OFF;
  }
}

static Boolean cpu_getCarry(CPU cpu) {
```

```
  assert(cpu != NULL);
  if ((cpu->status & MASK_STATUS_CARRY_ON) == MASK_STATUS_CARRY_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

// set to 0 - if a borrow is required
// set to 1 - if no borrow is required.
static void cpu_updateCarry_subtract(CPU cpu, Byte a, Byte b) {
  assert(cpu != NULL);

  // add a 'borrow' bit to borrow from
  Word aa = a;
  aa += MASK_BIT8;

  Word bb = b;

  Word result = aa-bb;

  if ((result & MASK_BIT8) == MASK_BIT8) {

    // subtraction didn't need to borrow from the 8th bit
    cpu_setCarry(cpu, TRUE);

  } else {

    // needed to borrow from the 8th bit
    cpu_setCarry(cpu, FALSE);
  }
}

// CMP - Compare
// This instruction compares the contents of the accumulator with another memory held
value and sets the zero and carry flags as appropriate.
static void cpu_CMP(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("CMP\n");

  Byte data = nes_readCPUMemory(nes, address);

  // C         Carry Flag     Set if A >= M

  cpu_updateCarry_subtract(cpu, cpu->accumulator, data);

  // Z  Zero Flag      Set if A = M

  cpu_updateZero(cpu, cpu->accumulator - data);

  // N  Negative Flag  Set if bit 7 of the result is set

  cpu_updateNegative(cpu, cpu->accumulator - data);
}

// CPX - Compare X Register
// This instruction compares the contents of the X register with another memory held
value and sets the zero and carry flags as appropriate.
static void cpu_CPX(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("CPX\n");

  Byte data = nes_readCPUMemory(nes, address);

  // C         Carry Flag     Set if X >= M
```

```
  cpu_updateCarry_subtract(cpu, cpu->indexX, data);

  // Z  Zero Flag      Set if X = M

  cpu_updateZero(cpu, cpu->indexX - data);

  // N  Negative Flag  Set if bit 7 of the result is set

  cpu_updateNegative(cpu, cpu->indexX - data);
}

// CPY - Compare Y Register
// This instruction compares the contents of the Y register with another memory held
value and sets the zero and carry flags as appropriate.
static void cpu_CPY(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("CPY\n");

  Byte data = nes_readCPUMemory(nes, address);

  // C         Carry Flag     Set if Y >= M

  cpu_updateCarry_subtract(cpu, cpu->indexY, data);

  // Z  Zero Flag      Set if Y = M

  cpu_updateZero(cpu, cpu->indexY - data);

  // N  Negative Flag  Set if bit 7 of the result is set

  cpu_updateNegative(cpu, cpu->indexY - data);
}

// this looks correct to me, but doesn't work on one of the test cases, so i use a
different method now
static void cpu_updateOverflow(CPU cpu, Byte a, Byte b, Byte c) {
  assert(cpu != NULL);

              // positive                    // positive                     // negative
  if ( ((a & MASK_BIT7) == MASK_BIT7) && ((b & MASK_BIT7) == MASK_BIT7) && ((c &
MASK_BIT7) == 0) ) {

    cpu_setOverflow(cpu, TRUE);

              // negative                 // negative                 // positive
 } else if ( ((a & MASK_BIT7) == 0) && ((b & MASK_BIT7) == 0) && ((c & MASK_BIT7) ==
MASK_BIT7) ) {

    cpu_setOverflow(cpu, TRUE);

  } else {
    cpu_setOverflow(cpu, FALSE);
  }
}

// ADC - Add with Carry
// This instruction adds the contents of a memory location to the accumulator together
with the carry bit.
// If overflow occurs the carry bit is set, this enables multiple byte addition to be
performed.
static void cpu_ADC(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("ADC\n");

  // A + M + C
```

```
  Byte data = nes_readCPUMemory(nes, address);

  Word result = cpu->accumulator + data;

  if (cpu_getCarry(cpu) == TRUE) {
    result++;
  }

  // C  Carry Flag     Set if overflow in bit 7

  if ((result & MASK_BIT8) == MASK_BIT8) {
    cpu_setCarry(cpu, TRUE);
  } else {
    cpu_setCarry(cpu, FALSE);
  }

  // V  Overflow Flag  Set if sign bit is incorrect

  // based on  http://nesdev.parodius.com/6502.txt
  if (((cpu->accumulator ^ result) & 0x80) && !((cpu->accumulator ^ data) & 0x80)) {
    cpu_setOverflow(cpu, TRUE);
  } else {
    cpu_setOverflow(cpu, FALSE);
  }

  cpu->accumulator = result;

  // Z  Zero Flag      Set if A = 0

  cpu_updateZero(cpu, cpu->accumulator);

  // N  Negative Flag  Set if bit 7 set

  cpu_updateNegative(cpu, cpu->accumulator);
}

// SBC - Subtract with Carry
// This instruction subtracts the contents of a memory location to the accumulator
together with the not of the carry bit.
// If overflow occurs the carry bit is clear, this enables multiple byte subtraction to
be performed.
static void cpu_SBC(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("SBC\n");

  // A,Z,C,N = A-M-(1-C)

  Byte data = nes_readCPUMemory(nes, address);

  Word result = cpu->accumulator - data;

  if (cpu_getCarry(cpu) == FALSE) {
    result--;
  }

  // C         Carry Flag     Clear if overflow in bit 7

  if ((result & MASK_BIT8) == MASK_BIT8) {
    cpu_setCarry(cpu, FALSE);
  } else {
    cpu_setCarry(cpu, TRUE);
  }

  // V  Overflow Flag  Set if sign bit is incorrect

  // based on http://nesdev.parodius.com/6502.txt
  if (((cpu->accumulator ^ result) & 0x80) && ((cpu->accumulator ^ data) & 0x80)) {
    cpu_setOverflow(cpu, TRUE);
  } else {
```

```
    cpu_setOverflow(cpu, FALSE);
  }

  cpu->accumulator = result;

  // Z  Zero Flag      Set if A = 0

  cpu_updateZero(cpu, cpu->accumulator);

  // N  Negative Flag  Set if bit 7 set

  cpu_updateNegative(cpu, cpu->accumulator);
}

// ASL - Arithmetic Shift Left

// This operation shifts all the bits of the memory contents one bit left.
// Bit 0 is set to 0 and bit 7 is placed in the carry flag.

// The effect of this operation is to multiply the memory contents by 2 (ignoring 2's
complement considerations),
// setting the carry if the result will not fit in 8 bits.
static void cpu_ASL_memory(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("ASL_memory\n");

  // A,Z,C,N = M*2 or M,Z,C,N = M*2

  Byte data = nes_readCPUMemory(nes, address);

  // dummy write, cycle wasted here
  nes_writeCPUMemory(nes, address, data);

  Word result = data << 1;

  // C Carry Flag      Set to contents of old bit 7

  if ( (data & MASK_BIT7) == MASK_BIT7) {
    cpu_setCarry(cpu, TRUE);
  } else {
    cpu_setCarry(cpu, FALSE);
  }

  // Z  Zero Flag      Set if A = 0

  cpu_updateZero(cpu, result);

  // N  Negative Flag  Set if bit 7 of the result is set

  cpu_updateNegative(cpu, result);

  nes_writeCPUMemory(nes, address, result);
}

// LSR - Logical Shift Right
// Each of the bits in A or M is shift one place to the right.
// The bit that was in bit 0 is shifted into the carry flag.
// Bit 7 is set to zero.
static void cpu_LSR_memory(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("LSR_memory\n");

  // A,C,Z,N = A/2 or M,C,Z,N = M/2

  Byte data = nes_readCPUMemory(nes, address);
```

```
  // dummy write, cycle wasted here
  nes_writeCPUMemory(nes, address, data);

  Word result = data >> 1;

  // C         Carry Flag     Set to contents of old bit 0

  if ( (data & MASK_BIT0) == MASK_BIT0) {
    cpu_setCarry(cpu, TRUE);
  } else {
    cpu_setCarry(cpu, FALSE);
  }

  // Z  Zero Flag      Set if result = 0

  cpu_updateZero(cpu, result);

  // N  Negative Flag  Set if bit 7 of the result is set

  cpu_updateNegative(cpu, result);

  nes_writeCPUMemory(nes, address, result);
}

// ROL - Rotate Left
// Move each of the bits in either A or M one place to the left.
// Bit 0 is filled with the current value of the carry flag whilst the old bit 7 becomes
the new carry flag value.
static void cpu_ROL_memory(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("ROL_memory\n");

  Byte data = nes_readCPUMemory(nes, address);

  // dummy write, cycle wasted here
  nes_writeCPUMemory(nes, address, data);

  Word result  = data << 1;

  if (cpu_getCarry(cpu) == TRUE) {
    result |= MASK_BIT0;
  }

  // C         Carry Flag     Set to contents of old bit 7

  if ((data & MASK_BIT7) == MASK_BIT7) {
    cpu_setCarry(cpu, TRUE);
  } else {
    cpu_setCarry(cpu, FALSE);
  }

  // Z  Zero Flag      Set if A = 0

  cpu_updateZero(cpu, result);

  // N  Negative Flag  Set if bit 7 of the result is set

  cpu_updateNegative(cpu, result);

  nes_writeCPUMemory(nes, address, result);

}

// ROR - Rotate Right
// Move each of the bits in either A or M one place to the right.
// Bit 7 is filled with the current value of the carry flag whilst the old bit 0 becomes
the new carry flag value.
static void cpu_ROR_memory(NES nes, Address address) {
  assert(nes != NULL);
```

```
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("ROR_memory\n");

  Byte data = nes_readCPUMemory(nes, address);

  // dummy write, cycle wasted here
  nes_writeCPUMemory(nes, address, data);

  Word result = data >> 1;

  if (cpu_getCarry(cpu) == TRUE) {
    result |= MASK_BIT7;
  }

  // C          Carry Flag     Set to contents of old bit 0

  if ((data & MASK_BIT0) == MASK_BIT0) {
    cpu_setCarry(cpu, TRUE);
  } else {
    cpu_setCarry(cpu, FALSE);
  }

  // Z  Zero Flag      Set if A = 0

  cpu_updateZero(cpu, result);

  // N  Negative Flag  Set if bit 7 of the result is set

  cpu_updateZero(cpu, result);

  nes_writeCPUMemory(nes, address, result);
}

// INC - Increment Memory
// Adds one to the value held at a specified memory location setting the zero and
negative flags as appropriate.
static void cpu_INC(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("INC\n");

  Byte data = nes_readCPUMemory(nes, address);

  // dummy write, cycle wasted here
  nes_writeCPUMemory(nes, address, data);

  data++;
  // Z  Zero Flag      Set if result is zero

  cpu_updateZero(cpu, data);

  // N  Negative Flag  Set if bit 7 of the result is set

  cpu_updateNegative(cpu, data);

  nes_writeCPUMemory(nes, address, data);
}

// DEC - Decrement Memory
// Subtracts one from the value held at a specified memory location setting the zero and
negative flags as appropriate.
static void cpu_DEC(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("DEC\n");
```

```
  Byte data = nes_readCPUMemory(nes, address);

  // dummy write, cycle wasted here
  nes_writeCPUMemory(nes, address, data);

  data--;

  // Z  Zero Flag      Set if result is zero

  cpu_updateZero(cpu, data);

  // N  Negative Flag  Set if bit 7 of the result is set

  cpu_updateNegative(cpu, data);

  nes_writeCPUMemory(nes, address, data);
}

// The processor supports a 256 byte stack located between $0100 and $01FF.
// The stack is located at memory locations $0100-$01FF.
// The stack pointer is an 8-bit register which serves as an offset from $0100.
// The stack pointer is an 8 bit register and holds the low 8 bits of the next free
location on the stack.
// Pushing bytes to the stack causes the stack pointer to be decremented. Conversely
pulling bytes causes it to be incremented.
// There is no detection of stack overflow and the stack pointer will just wrap around
from $00 to $FF.

static void cpu_pushStack(NES nes, Byte data) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  Address stackAddress = GET_STACK_ADDRESS(cpu->stackPointer);
  VALIDATE_STACK_ADDRESS(stackAddress);

  nes_writeCPUMemory(nes, stackAddress, data);

  cpu->stackPointer--;

  VALIDATE_STACK_POINTER(cpu->stackPointer);
}

static Byte cpu_popStack(NES nes) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  VALIDATE_STACK_POINTER(cpu->stackPointer);
  cpu->stackPointer++;
  VALIDATE_STACK_POINTER(cpu->stackPointer);

  Address stackAddress = GET_STACK_ADDRESS(cpu->stackPointer);
  VALIDATE_STACK_ADDRESS(stackAddress);

  Byte data = nes_readCPUMemory(nes, stackAddress);

  return data;
}

// JSR - Jump to Subroutine
// The JSR instruction pushes the address (minus one) of the return point on to the
stack and then sets the program counter to the target memory address.
static void cpu_JSR(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("JSR\n");

  // During a JSR, the address pushed onto the stack is that of the 3rd byte of the
instruction - that is, 1 byte BEFORE the next instruction.
```

```
  // This is because it pushes the program counter onto the stack BEFORE it fetches the
final byte of the opcode
  // (and, as such, before it can increment the PC past this point).

  // To compensate for this, the RTS opcode increments the program counter during its
6th instruction cycle.

  // JSR takes 6 cycles, waste a cycle here
  nes_cpuCycled(nes);

  Address returnAddress = cpu->programCounter - 1;

  cpu_pushStack(nes, GET_ADDRESS_HIGH_BYTE(returnAddress));
  cpu_pushStack(nes, GET_ADDRESS_LOW_BYTE(returnAddress));

  cpu->programCounter = address;
}

// ASL - Arithmetic Shift Left
// This operation shifts all the bits of the accumulator one bit left.
// Bit 0 is set to 0 and bit 7 is placed in the carry flag.

// The effect of this operation is to multiply the memory contents by 2
// (ignoring 2's complement considerations), setting the carry if the result will not
fit in 8 bits.
static void cpu_ASL(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("ASL\n");

  Word result = cpu->accumulator << 1;

  // C         Carry Flag     Set to contents of old bit 7

  if ((cpu->accumulator & MASK_BIT7) == MASK_BIT7) {
    cpu_setCarry(cpu, TRUE);
  } else {
    cpu_setCarry(cpu, FALSE);
  }

  cpu->accumulator = result;

  // Z  Zero Flag      Set if A = 0

  cpu_updateZero(cpu, cpu->accumulator);

  // N  Negative Flag  Set if bit 7 of the result is set

  cpu_updateNegative(cpu, cpu->accumulator);

}

// LSR - Logical Shift Right
// Each of the bits in A is shift one place to the right.
// The bit that was in bit 0 is shifted into the carry flag. Bit 7 is set to zero.
static void cpu_LSR(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("LSR\n");

  Word result = cpu->accumulator >> 1;

  // C         Carry Flag     Set to contents of old bit 0

  if ( (cpu->accumulator & MASK_BIT0) == MASK_BIT0 ) {
    cpu_setCarry(cpu, TRUE);
  } else {
    cpu_setCarry(cpu, FALSE);
  }

  cpu->accumulator = result;
```

```
  // Z  Zero Flag      Set if result = 0

  cpu_updateZero(cpu, cpu->accumulator);

  // N  Negative Flag  Set if bit 7 of the result is set

  cpu_updateNegative(cpu, cpu->accumulator);
}

// ROL - Rotate Left
// Move each of the bits in either A or M one place to the left.
// Bit 0 is filled with the current value of the carry flag whilst the old bit 7 becomes
the new carry flag value.
static void cpu_ROL(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("ROL\n");

  Word result = cpu->accumulator << 1;

  if (cpu_getCarry(cpu) == TRUE) {
    result |= MASK_BIT0;
  }

  if ( (cpu->accumulator & MASK_BIT7) == MASK_BIT7) {
    cpu_setCarry(cpu, TRUE);
  } else {
    cpu_setCarry(cpu, FALSE);
  }

  // Z  Zero Flag      Set if A = 0

  cpu->accumulator = result;

  cpu_updateZero(cpu, cpu->accumulator);

  // N  Negative Flag  Set if bit 7 of the result is set

  cpu_updateNegative(cpu, cpu->accumulator);
}

// ROR - Rotate Right
// Move each of the bits in either A or M one place to the right.
// Bit 7 is filled with the current value of the carry flag whilst the old bit 0 becomes
the new carry flag value.
static void cpu_ROR(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("ROR\n");

  Word result = cpu->accumulator >> 1;

  if (cpu_getCarry(cpu) == TRUE) {
    result |= MASK_BIT7;
  }

  // C         Carry Flag     Set to contents of old bit 0

  if ( (cpu->accumulator & MASK_BIT0) == MASK_BIT0) {
    cpu_setCarry(cpu, TRUE);
  } else {
    cpu_setCarry(cpu, FALSE);
  }

  cpu->accumulator = result;

  // Z  Zero Flag      Set if A = 0

  cpu_updateZero(cpu, cpu->accumulator);

  // N  Negative Flag  Set if bit 7 of the result is set
```

```
  cpu_updateNegative(cpu, cpu->accumulator);
}

static void cpu_setInterruptDisable(CPU cpu, Boolean state) {
  assert(cpu != NULL);

  if (state == TRUE) {
    cpu->status |= MASK_STATUS_INTERRUPT_ON;
  } else if (state == FALSE) {
    cpu->status &= MASK_STATUS_INTERRUPT_OFF;
  }
}

Boolean cpu_getInterruptDisable(CPU cpu) {
  assert(cpu != NULL);

  if ((cpu->status & MASK_STATUS_INTERRUPT_ON) == MASK_STATUS_INTERRUPT_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void cpu_setBreak(CPU cpu, Boolean state) {
  assert(cpu != NULL);

  if (state == TRUE) {
    cpu->status |= MASK_STATUS_BREAK_ON;
  } else if (state == FALSE) {
    cpu->status &= MASK_STATUS_BREAK_OFF;
  }
}

static Boolean cpu_getBreak(CPU cpu) {
  assert(cpu != NULL);

  if ( (cpu->status & MASK_STATUS_BREAK_ON) == MASK_STATUS_BREAK_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

void cpu_handleInterrupt(NES nes, Address handlerLowByte, Boolean fromBRK) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("handleInterrupt\n");

  cpu_setInterruptDisable(cpu, TRUE);

  if (fromBRK == TRUE) {
    cpu_setBreak(cpu, TRUE);
  } else {
    cpu_setBreak(cpu, FALSE);
  }

  cpu_pushStack(nes, GET_ADDRESS_HIGH_BYTE(cpu->programCounter));

  cpu_pushStack(nes, GET_ADDRESS_LOW_BYTE(cpu->programCounter));

  // 1 at all times
  cpu->status |= MASK_BIT5;

  cpu_pushStack(nes, cpu->status);

  Address address = nes_readCPUMemory(nes, handlerLowByte);
  address += nes_readCPUMemory(nes, handlerLowByte+1) << BITS_PER_BYTE;

  debug_printf("Address is %d\n", address);
```

```
  cpu->programCounter = address;

}

// BRK - Force Interrupt
// The BRK instruction forces the generation of an interrupt request.

// The program counter and processor status are pushed on the stack
// then the IRQ interrupt vector at $FFFE/F is loaded into the PC
// and the break flag in the status set to one.
static void cpu_BRK(NES nes) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("BRK\n");

  // the instruction has a "useless" second byte we need to skip
  // (added for debugging purposes apparently)
  cpu->programCounter++;

  // B         Break Command          Set to 1
  cpu_handleInterrupt(nes, CPU_IRQ_VECTOR_LOWER_ADDRESS, TRUE);
}

// RTS - Return from Subroutine
// The RTS instruction is used at the end of a subroutine to return to the calling
routine.
// It pulls the program counter (minus one) from the stack.
static void cpu_RTS(NES nes) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("RTS\n");

  // waste a cycle for the stack pointer increment that is about to happen in popStack
  nes_cpuCycled(nes);

  Address address = cpu_popStack(nes);
  address += cpu_popStack(nes) << BITS_PER_BYTE;

  // waste a cycle to increment the program counter
  nes_cpuCycled(nes);

  address++;

  cpu->programCounter = address;
}

// RTI - Return from Interrupt
// The RTI instruction is used at the end of an interrupt processing routine.
// It pulls the processor flags from the stack followed by the program counter.
static void cpu_RTI(NES nes) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("RTI\n");

  // waste a cycle for the initial stack pointer increment that is about to happen in
popStack
  nes_cpuCycled(nes);

  cpu->status = cpu_popStack(nes);

  // 1 at all times
  cpu->status |= MASK_BIT5;

  Address address = cpu_popStack(nes);
  address += cpu_popStack(nes) << BITS_PER_BYTE;
```

```
  cpu->programCounter = address;
}

// PHP - Push Processor Status
// Pushes a copy of the status flags on to the stack.
static void cpu_PHP(NES nes) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("PHP\n");

  // The status bits pushed on the stack by PHP have the breakpoint bit set.

  // 1 at all times
  cpu->status |= MASK_BIT5;

  Byte data = cpu->status;
  data |= MASK_STATUS_BREAK_ON;

  //printf("PHP 0x%x 0x%x\n", cpu->status, data);

  cpu_pushStack(nes, data);
}

// PLP - Pull Processor Status
// Pulls an 8 bit value from the stack and into the processor flags.
// The flags will take on new states as determined by the value pulled.
static void cpu_PLP(NES nes) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("PLP\n");

  // waste a cycle for the stack pointer increment that is about to happen in popStack
  nes_cpuCycled(nes);

  Byte data = cpu_popStack(nes);

  //printf("PLP 0x%x 0x%x\n", cpu->status, data);

  cpu->status = data;

  // 1 at all times
  cpu->status |= MASK_BIT5;
}

// PHA - Push Accumulator
// Pushes a copy of the accumulator on to the stack.
static void cpu_PHA(NES nes) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("PHA\n");

  cpu_pushStack(nes, cpu->accumulator);
}

// PLA - Pull Accumulator
// Pulls an 8 bit value from the stack and into the accumulator.
// The zero and negative flags are set as appropriate.
static void cpu_PLA(NES nes) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("PLA\n");

  // waste a cycle for the stack pointer increment that is about to happen in popStack
```

```
  nes_cpuCycled(nes);

  cpu->accumulator = cpu_popStack(nes);

  // Z  Zero Flag      Set if A = 0

  cpu_updateZero(cpu, cpu->accumulator);

  // N  Negative Flag  Set if bit 7 of A is set
  cpu_updateNegative(cpu, cpu->accumulator);
}

// INX - Increment X Register
// Adds one to the X register setting the zero and negative flags as appropriate.
static void cpu_INX(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("INX\n");

  cpu->indexX++;

  // Z  Zero Flag      Set if X is zero

  cpu_updateZero(cpu, cpu->indexX);

  // N  Negative Flag  Set if bit 7 of X is set

  cpu_updateNegative(cpu, cpu->indexX);
}

// DEX - Decrement X Register
// Subtracts one from the X register setting the zero and negative flags as appropriate.
static void cpu_DEX(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("DEX\n");

  cpu->indexX--;

  // Z  Zero Flag      Set if X is zero

  cpu_updateZero(cpu, cpu->indexX);

  // N  Negative Flag  Set if bit 7 of X is set

  cpu_updateNegative(cpu, cpu->indexX);
}

// INY - Increment Y Register
// Adds one to the Y register setting the zero and negative flags as appropriate.
static void cpu_INY(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("INY\n");

  cpu->indexY++;

  // Z  Zero Flag      Set if Y is zero

  cpu_updateZero(cpu, cpu->indexY);

  // N  Negative Flag  Set if bit 7 of Y is set

  cpu_updateNegative(cpu, cpu->indexY);
}

// DEY - Decrement Y Register
// Subtracts one from the Y register setting the zero and negative flags as appropriate.
static void cpu_DEY(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("DEY\n");
```

```
  cpu->indexY--;

  // Z  Zero Flag      Set if Y is zero

  cpu_updateZero(cpu, cpu->indexY);

  // N  Negative Flag  Set if bit 7 of Y is set
  cpu_updateNegative(cpu, cpu->indexY);
}

// TAX - Transfer Accumulator to X
// Copies the current contents of the accumulator into the X register and sets the zero
and negative flags as appropriate.
static void cpu_TAX(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("TAX\n");

  cpu->indexX = cpu->accumulator;

  // Z  Zero Flag      Set if X = 0

  cpu_updateZero(cpu, cpu->indexX);

  // N  Negative Flag  Set if bit 7 of X is set

  cpu_updateNegative(cpu, cpu->indexX);
}

// TXA - Transfer X to Accumulator
// Copies the current contents of the X register into the accumulator and sets the zero
and negative flags as appropriate.
static void cpu_TXA(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("TXA\n");

  cpu->accumulator = cpu->indexX;

  // Z  Zero Flag      Set if A = 0

  cpu_updateZero(cpu, cpu->accumulator);

  // N  Negative Flag  Set if bit 7 of A is set
  cpu_updateNegative(cpu, cpu->accumulator);
}

// TYA - Transfer Y to Accumulator
// Copies the current contents of the Y register into the accumulator and sets the zero
and negative flags as appropriate.
static void cpu_TYA(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("TYA\n");

  cpu->accumulator = cpu->indexY;

  // Z  Zero Flag      Set if A = 0
  cpu_updateZero(cpu, cpu->accumulator);

  // N  Negative Flag  Set if bit 7 of A is set
  cpu_updateNegative(cpu, cpu->accumulator);
}

// TSX - Transfer Stack Pointer to X
// Copies the current contents of the stack register into the X register and sets the
zero and negative flags as appropriate.
static void cpu_TSX(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("TSX\n");
```

```
  cpu->indexX = cpu->stackPointer;

  // Z  Zero Flag      Set if X = 0

  cpu_updateZero(cpu, cpu->indexX);

  // N  Negative Flag  Set if bit 7 of X is set

  cpu_updateNegative(cpu, cpu->indexX);
}


// TAY - Transfer Accumulator to Y
// Copies the current contents of the accumulator into the Y register and sets the zero
and negative flags as appropriate.
static void cpu_TAY(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("TAY\n");

  cpu->indexY = cpu->accumulator;

  // Z  Zero Flag      Set if Y = 0

  cpu_updateZero(cpu, cpu->indexY);

  // N  Negative Flag  Set if bit 7 of Y is set
  cpu_updateNegative(cpu, cpu->indexY);
}

// TXS - Transfer X to Stack Pointer
// Copies the current contents of the X register into the stack register.
static void cpu_TXS(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("TXS\n");

  cpu->stackPointer = cpu->indexX;
}

// SEI - Set Interrupt Disable
static void cpu_SEI(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("SEI\n");

  // I         Interrupt Disable      Set to 1

  cpu_setInterruptDisable(cpu, TRUE);
}

// CLI - Clear Interrupt Disable
static void cpu_CLI(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("CLI\n");

  // I         Interrupt Disable      Set to 0

  cpu_setInterruptDisable(cpu, FALSE);
}

// SEC - Set Carry Flag
static void cpu_SEC(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("SEC\n");

  cpu_setCarry(cpu, TRUE);
}
```

```
// CLC - Clear Carry Flag
static void cpu_CLC(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("CLC\n");

  // C          Carry Flag     Set to 0

  cpu_setCarry(cpu, FALSE);
}

// CLV - Clear Overflow Flag
static void cpu_CLV(CPU cpu) {
  assert(cpu != NULL);

  debug_printf("CLV\n");

  // V          Overflow Flag          Set to 0

  cpu_setOverflow(cpu, FALSE);
}

// BEQ - Branch if Equal
// If the zero flag is set then add the relative displacement to the program counter to
cause a branch to a new location.
static void cpu_BEQ(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("BEQ\n");

  SignedByte data = nes_readCPUMemory(nes, address);

  if (cpu_getZero(cpu) == TRUE) {

    // +1 cycle if the branch succeeds
    nes_cpuCycled(nes);

    cpu->programCounter += data;
  }
}

// BNE - Branch if Not Equal
// If the zero flag is clear then add the relative displacement to the program counter
to cause a branch to a new location.
static void cpu_BNE(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("BNE\n");

  SignedByte data = nes_readCPUMemory(nes, address);

  if (cpu_getZero(cpu) == FALSE) {

    // +1 cycle if the branch succeeds
    nes_cpuCycled(nes);

    cpu->programCounter += data;
  }
}


// BMI - Branch if Minus
// If the negative flag is set then add the relative displacement to the program counter
to cause a branch to a new location.
static void cpu_BMI(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);
```

```
  debug_printf("BMI\n");

  SignedByte data = nes_readCPUMemory(nes, address);

  if (cpu_getNegative(cpu) == TRUE) {

    // +1 cycle if the branch succeeds
    nes_cpuCycled(nes);

    cpu->programCounter += data;

  }
}

// BPL - Branch if Positive
// If the negative flag is clear then add the relative displacement to the program
counter to cause a branch to a new location.
static void cpu_BPL(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("BPL\n");

  SignedByte data = nes_readCPUMemory(nes, address);

  if (cpu_getNegative(cpu) == FALSE) {

    // +1 cycle if the branch succeeds
    nes_cpuCycled(nes);

    cpu->programCounter += data;
  }
}

// BCC - Branch if Carry Clear
// If the carry flag is clear then add the relative displacement to the program counter
to cause a branch to a new location.
static void cpu_BCC(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("BCC\n");

  SignedByte data = nes_readCPUMemory(nes, address);

  if (cpu_getCarry(cpu) == FALSE) {

    // +1 cycle if the branch succeeds
    nes_cpuCycled(nes);

    cpu->programCounter += data;
  }
}

// BCS - Branch if Carry Set
// If the carry flag is set then add the relative displacement to the program counter to
cause a branch to a new location.
static void cpu_BCS(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("BCS\n");

  SignedByte data = nes_readCPUMemory(nes, address);

  if (cpu_getCarry(cpu) == TRUE) {

    // +1 cycle if the branch succeeds
```

```
    nes_cpuCycled(nes);

    cpu->programCounter += data;
  }
}

// BVC - Branch if Overflow Clear
// If the overflow flag is clear then add the relative displacement to the program
counter to cause a branch to a new location.
static void cpu_BVC(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("BVC\n");

  SignedByte data = nes_readCPUMemory(nes, address);

  if (cpu_getOverflow(cpu) == FALSE) {

    // +1 cycle if the branch succeeds
    nes_cpuCycled(nes);

    cpu->programCounter += data;
  }
}

// BVS - Branch if Overflow Set
// If the overflow flag is set then add the relative displacement to the program counter
to cause a branch to a new location.
static void cpu_BVS(NES nes, Address address) {
  assert(nes != NULL);
  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  debug_printf("BVS\n");

  SignedByte data = nes_readCPUMemory(nes, address);

  if (cpu_getOverflow(cpu) == TRUE) {

    // +1 cycle if the branch succeeds
    nes_cpuCycled(nes);

    cpu->programCounter += data;
  }
}

void cpu_step(NES nes) {
  assert(nes != NULL);

  debug_printf("cpu_step\n");

  CPU cpu = nes_getCPU(nes);
  assert(cpu != NULL);

  Byte instruction = nes_readCPUMemory(nes, cpu->programCounter);

  cpu_increaseProgramCounter(cpu);

  debug_printf("Instruction: 0x%x\n", instruction);

  Address address = 0;

  switch(instruction) {

    case LDA_IMM:
    case LDX_IMM:
    case LDY_IMM:
    case AND_IMM:
    case ORA_IMM:
    case EOR_IMM:
```

```
case CMP_IMM:
case CPX_IMM:
case CPY_IMM:
case ADC_IMM:
case SBC_IMM:

  address = cpu->programCounter;
  cpu_increaseProgramCounter(cpu);

  break;

case LDA_ZPAGE:
case STA_ZPAGE:
case LDX_ZPAGE:
case STX_ZPAGE:
case LDY_ZPAGE:
case STY_ZPAGE:
case AND_ZPAGE:
case ORA_ZPAGE:
case EOR_ZPAGE:
case BIT_ZPAGE:
case CMP_ZPAGE:
case CPX_ZPAGE:
case CPY_ZPAGE:
case ADC_ZPAGE:
case SBC_ZPAGE:
case ASL_ZPAGE:
case LSR_ZPAGE:
case ROL_ZPAGE:
case ROR_ZPAGE:
case INC_ZPAGE:
case DEC_ZPAGE:

  address = nes_readCPUMemory(nes, cpu->programCounter);
  cpu_increaseProgramCounter(cpu);

  break;

case LDA_ZPAGEX:
case STA_ZPAGEX:
case LDY_ZPAGEX:
case STY_ZPAGEX:
case AND_ZPAGEX:
case ORA_ZPAGEX:
case EOR_ZPAGEX:
case CMP_ZPAGEX:
case ADC_ZPAGEX:
case SBC_ZPAGEX:
case ASL_ZPAGEX:
case LSR_ZPAGEX:
case ROL_ZPAGEX:
case ROR_ZPAGEX:
case INC_ZPAGEX:
case DEC_ZPAGEX:

  {
    Byte data = nes_readCPUMemory(nes, cpu->programCounter);
    cpu_increaseProgramCounter(cpu);
    data += cpu->indexX;
    address = data;
  }

  break;


case LDA_ABS:
case STA_ABS:
case LDX_ABS:
case STX_ABS:
case LDY_ABS:
case STY_ABS:
case AND_ABS:
```

```
case ORA_ABS:
case EOR_ABS:
case BIT_ABS:
case CMP_ABS:
case CPX_ABS:
case CPY_ABS:
case ADC_ABS:
case SBC_ABS:
case ASL_ABS:
case LSR_ABS:
case ROL_ABS:
case ROR_ABS:
case INC_ABS:
case DEC_ABS:
case JSR_ABS:
case JMP_ABS:

  address = nes_readCPUMemory(nes, cpu->programCounter);
  cpu_increaseProgramCounter(cpu);

  address += nes_readCPUMemory(nes, cpu->programCounter) << BITS_PER_BYTE;
  cpu_increaseProgramCounter(cpu);

  break;

case LDA_ABSX:
case STA_ABSX:
case LDY_ABSX:
case AND_ABSX:
case ORA_ABSX:
case EOR_ABSX:
case CMP_ABSX:
case ADC_ABSX:
case SBC_ABSX:
case ASL_ABSX:
case LSR_ABSX:
case ROL_ABSX:
case ROR_ABSX:
case INC_ABSX:
case DEC_ABSX:

  address = nes_readCPUMemory(nes, cpu->programCounter);
  cpu_increaseProgramCounter(cpu);

  address += nes_readCPUMemory(nes, cpu->programCounter) << BITS_PER_BYTE;
  cpu_increaseProgramCounter(cpu);

  address += cpu->indexX;

  break;


case LDA_ABSY:
case STA_ABSY:
case LDX_ABSY:
case AND_ABSY:
case ORA_ABSY:
case EOR_ABSY:
case CMP_ABSY:
case ADC_ABSY:
case SBC_ABSY:

  address = nes_readCPUMemory(nes, cpu->programCounter);
  cpu_increaseProgramCounter(cpu);

  address += nes_readCPUMemory(nes, cpu->programCounter) << BITS_PER_BYTE;
  cpu_increaseProgramCounter(cpu);

  address += cpu->indexY;

  break;
```

```
    case LDA_INDX:
    case STA_INDX:
    case AND_INDX:
    case ORA_INDX:
    case EOR_INDX:
    case CMP_INDX:
    case ADC_INDX:
    case SBC_INDX:

      {
        Byte data = nes_readCPUMemory(nes, cpu->programCounter);
        cpu_increaseProgramCounter(cpu);

        data += cpu->indexX;

        Byte lowAddress = data;
        Byte highAddress = data + 1;

        address = nes_readCPUMemory(nes, lowAddress);
        address += nes_readCPUMemory(nes, highAddress) << BITS_PER_BYTE;
      }


      break;

    case LDA_INDY:
    case STA_INDY:
    case AND_INDY:
    case ORA_INDY:
    case EOR_INDY:
    case CMP_INDY:
    case ADC_INDY:
    case SBC_INDY:

      {
        Byte data = nes_readCPUMemory(nes, cpu->programCounter);
        cpu_increaseProgramCounter(cpu);

        Byte lowAddress = data;
        Byte highAddress = data+1;

        address = nes_readCPUMemory(nes, lowAddress);
        address += nes_readCPUMemory(nes, highAddress) << BITS_PER_BYTE;    // is the +1
meant to wraparound to zero page as well?

        address += cpu->indexY;
      }



      break;


    case LDX_ZPAGEY:
    case STX_ZPAGEY:

      {
        Byte data = nes_readCPUMemory(nes, cpu->programCounter);
        cpu_increaseProgramCounter(cpu);
        data += cpu->indexY;
        address = data;
      }

      break;

    case ASL_ACCUM:
    case LSR_ACCUM:
    case ROL_ACCUM:
    case ROR_ACCUM:

      // these take 2 cycles. do a dummy read so that the ppu/apu get to advance
      nes_cpuCycled(nes);
```

```
    break;

case JMP_INDIRECT_CODE1:
case JMP_INDIRECT_CODE2:

  {
    Byte directAddressLow = nes_readCPUMemory(nes, cpu->programCounter);
    cpu_increaseProgramCounter(cpu);

    Byte directAddressHigh = nes_readCPUMemory(nes, cpu->programCounter);
    cpu_increaseProgramCounter(cpu);

    Address lowAddress = directAddressLow;
    lowAddress += directAddressHigh << BITS_PER_BYTE;

    directAddressLow++;

    Address highAddress = directAddressLow;
    highAddress += directAddressHigh << BITS_PER_BYTE;

    address = nes_readCPUMemory(nes, lowAddress);
    address += nes_readCPUMemory(nes, highAddress) << BITS_PER_BYTE;
  }


  break;


case BRK:
case RTS:
case RTI:
case PHP:
case PLP:
case PHA:
case PLA:
case INX:
case DEX:
case INY:
case DEY:
case TAX:
case TXA:
case TAY:
case TYA:
case TSX:
case TXS:
case SED:
case CLD:
case SEI:
case CLI:
case SEC:
case CLC:
case CLV:

case NOP:

   // these take 2+ cycles. do a dummy read so that the ppu/apu get to advance
   nes_cpuCycled(nes);

  break;

case BEQ:
case BNE:
case BMI:
case BPL:
case BCC:
case BCS:
case BVC:
case BVS:

  address = cpu->programCounter;
  cpu_increaseProgramCounter(cpu);
```

```
    break;


  default:
    printf("Instruction not implemented: 0x%x\n", instruction);
    //debug_printf("Instruction not implemented: 0x%x\n", instruction);
    assert(FALSE);
}


switch(instruction) {

  case LDA_INDY:
  case LDA_IMM:
  case LDA_ZPAGE:
  case LDA_ZPAGEX:
  case LDA_ABS:
  case LDA_ABSX:
  case LDA_ABSY:
  case LDA_INDX:
    cpu_LDA(nes, address);
    break;

  case STA_INDY:
  case STA_ZPAGE:
  case STA_ZPAGEX:
  case STA_ABS:
  case STA_ABSX:
  case STA_ABSY:
  case STA_INDX:

    cpu_STA(nes, address);
    break;

  case LDX_IMM:
  case LDX_ZPAGE:
  case LDX_ABS:
  case LDX_ABSY:
  case LDX_ZPAGEY:
    cpu_LDX(nes, address);
    break;

  case STX_ZPAGE:
  case STX_ABS:
  case STX_ZPAGEY:
    cpu_STX(nes, address);
    break;

  case LDY_IMM:
  case LDY_ZPAGE:
  case LDY_ZPAGEX:
  case LDY_ABS:
  case LDY_ABSX:
    cpu_LDY(nes, address);
    break;

  case STY_ZPAGE:
  case STY_ZPAGEX:
  case STY_ABS:
    cpu_STY(nes, address);
    break;

  case AND_INDY:
  case AND_IMM:
  case AND_ZPAGE:
  case AND_ZPAGEX:
  case AND_ABS:
  case AND_ABSX:
  case AND_ABSY:
  case AND_INDX:
```

```
    cpu_AND(nes, address);
    break;

  case ORA_INDY:
  case ORA_IMM:
  case ORA_ZPAGE:
  case ORA_ZPAGEX:
  case ORA_ABS:
  case ORA_ABSX:
  case ORA_ABSY:
  case ORA_INDX:

    cpu_ORA(nes, address);
    break;

  case EOR_INDY:
  case EOR_IMM:
  case EOR_ZPAGE:
  case EOR_ZPAGEX:
  case EOR_ABS:
  case EOR_ABSX:
  case EOR_ABSY:
  case EOR_INDX:

    cpu_EOR(nes, address);
    break;

  case BIT_ZPAGE:
  case BIT_ABS:
    cpu_BIT(nes, address);
    break;

  case CMP_INDY:
  case CMP_IMM:
  case CMP_ZPAGE:
  case CMP_ZPAGEX:
  case CMP_ABS:
  case CMP_ABSX:
  case CMP_ABSY:
  case CMP_INDX:

    cpu_CMP(nes, address);
    break;

  case CPX_IMM:
  case CPX_ZPAGE:
  case CPX_ABS:
    cpu_CPX(nes, address);
    break;

  case CPY_IMM:
  case CPY_ZPAGE:
  case CPY_ABS:
    cpu_CPY(nes, address);
    break;

  case ADC_INDY:
  case ADC_IMM:
  case ADC_ZPAGE:
  case ADC_ZPAGEX:
  case ADC_ABS:
  case ADC_ABSX:
  case ADC_ABSY:
  case ADC_INDX:

    cpu_ADC(nes, address);
    break;

  case SBC_INDY:
  case SBC_IMM:
  case SBC_ZPAGE:
  case SBC_ZPAGEX:
```

```
case SBC_ABS:
case SBC_ABSX:
case SBC_ABSY:
case SBC_INDX:

  cpu_SBC(nes, address);
  break;

case ASL_ZPAGE:
case ASL_ZPAGEX:
case ASL_ABS:
case ASL_ABSX:
  cpu_ASL_memory(nes, address);
  break;

case LSR_ZPAGE:
case LSR_ZPAGEX:
case LSR_ABS:
case LSR_ABSX:
  cpu_LSR_memory(nes, address);
  break;

case ROL_ZPAGE:
case ROL_ZPAGEX:
case ROL_ABS:
case ROL_ABSX:
  cpu_ROL_memory(nes, address);
  break;

case ROR_ZPAGE:
case ROR_ZPAGEX:
case ROR_ABS:
case ROR_ABSX:
  cpu_ROR_memory(nes, address);
  break;

case INC_ZPAGE:
case INC_ZPAGEX:
case INC_ABS:
case INC_ABSX:
  cpu_INC(nes, address);
  break;

case DEC_ZPAGE:
case DEC_ZPAGEX:
case DEC_ABS:
case DEC_ABSX:
  cpu_DEC(nes, address);
  break;

case JSR_ABS:
  cpu_JSR(nes, address);
  break;

case JMP_ABS:
  cpu_JMP(nes, address);
  break;

case ASL_ACCUM:
  cpu_ASL(cpu);
  break;

case LSR_ACCUM:
  cpu_LSR(cpu);
  break;

case ROL_ACCUM:
  cpu_ROL(cpu);
  break;

case ROR_ACCUM:
  cpu_ROR(cpu);
```

```
    break;

  case JMP_INDIRECT_CODE1:
  case JMP_INDIRECT_CODE2:

    cpu_JMP(nes, address);
    break;

  case BRK:
    cpu_BRK(nes);
    break;

  case RTS:
    cpu_RTS(nes);
    break;

  case RTI:
    cpu_RTI(nes);
    break;

  case PHP:
    cpu_PHP(nes);
    break;

  case PLP:
    cpu_PLP(nes);
    break;

  case PHA:
    cpu_PHA(nes);
    break;

  case PLA:
    cpu_PLA(nes);
    break;

  case INX:
    cpu_INX(cpu);
    break;

  case DEX:
    cpu_DEX(cpu);
    break;

  case INY:
    cpu_INY(cpu);
    break;

  case DEY:
    cpu_DEY(cpu);
    break;

  case TAX:
    cpu_TAX(cpu);
    break;

  case TXA:
    cpu_TXA(cpu);
    break;

  case TAY:
    cpu_TAY(cpu);
    break;

  case TYA:
    cpu_TYA(cpu);
    break;

  case TSX:
    cpu_TSX(cpu);
    break;
```

```
  case TXS:
    cpu_TXS(cpu);
    break;

  case SED:
    cpu_SED(cpu);
    break;

  case CLD:
    cpu_CLD(cpu);
    break;

  case SEI:
    cpu_SEI(cpu);
    break;

  case CLI:
    cpu_CLI(cpu);
    break;

  case SEC:
    cpu_SEC(cpu);
    break;

  case CLC:
    cpu_CLC(cpu);
    break;

  case CLV:
    cpu_CLV(cpu);
    break;

  case BEQ:
    cpu_BEQ(nes, address);
    break;

  case BNE:
    cpu_BNE(nes, address);
    break;

  case BMI:
    cpu_BMI(nes, address);
    break;

  case BPL:
    cpu_BPL(nes, address);
    break;

  case BCC:
    cpu_BCC(nes, address);
    break;

  case BCS:
    cpu_BCS(nes, address);
    break;

  case BVC:
    cpu_BVC(nes, address);
    break;

  case BVS:
    cpu_BVS(nes, address);
    break;

  case NOP:
    break;

  default:
    printf("Instruction not implemented: 0x%x\n", instruction);
    //debug_printf("Instruction not implemented: 0x%x\n", instruction);
    assert(FALSE);
}
```

```
  debug_printf("Address: 0x%0x\n", address);

}

void cpu_tests(void) {

  {

    CPU cpu = cpu_init();
    assert(cpu != NULL);

    cpu_updateZero(cpu, 0);
    assert(cpu_getZero(cpu) == TRUE);
    assert(cpu_getNegative(cpu) == FALSE);
    assert(cpu_getOverflow(cpu) == FALSE);
    assert(cpu_getCarry(cpu) == FALSE);

    cpu_updateZero(cpu, 1);
    assert(cpu_getZero(cpu) == FALSE);
    assert(cpu_getNegative(cpu) == FALSE);
    assert(cpu_getOverflow(cpu) == FALSE);
    assert(cpu_getCarry(cpu) == FALSE);

  }

  {

    CPU cpu = cpu_init();
    assert(cpu != NULL);

    cpu_updateNegative(cpu, 128);
    assert(cpu_getNegative(cpu) == TRUE);
    assert(cpu_getOverflow(cpu) == FALSE);
    assert(cpu_getCarry(cpu) == FALSE);

    cpu_updateNegative(cpu, 127);
    assert(cpu_getNegative(cpu) == FALSE);
    assert(cpu_getOverflow(cpu) == FALSE);
    assert(cpu_getCarry(cpu) == FALSE);

    cpu_updateNegative(cpu, 0);
    assert(cpu_getNegative(cpu) == FALSE);
    assert(cpu_getOverflow(cpu) == FALSE);
    assert(cpu_getCarry(cpu) == FALSE);

    int i;
    for (i=0; i < 128; i++) {
      cpu_updateNegative(cpu, i);
      assert(cpu_getNegative(cpu) == FALSE);
      assert(cpu_getOverflow(cpu) == FALSE);
      assert(cpu_getCarry(cpu) == FALSE);
    }

    for (i=128; i < 256; i++) {
      cpu_updateNegative(cpu, i);
      assert(cpu_getNegative(cpu) == TRUE);
      assert(cpu_getOverflow(cpu) == FALSE);
      assert(cpu_getCarry(cpu) == FALSE);
    }

  }

  {

    CPU cpu = cpu_init();
    assert(cpu != NULL);

    cpu_setCarry(cpu, TRUE);
    assert(cpu_getCarry(cpu) == TRUE);
    assert(cpu_getOverflow(cpu) == FALSE);
```

```
  cpu_setCarry(cpu, FALSE);
  assert(cpu_getCarry(cpu) == FALSE);
  assert(cpu_getOverflow(cpu) == FALSE);

  cpu_updateCarry_subtract(cpu, 0, 0);
  assert(cpu_getCarry(cpu) == TRUE);

  cpu_updateCarry_subtract(cpu, 1, 0);
  assert(cpu_getCarry(cpu) == TRUE);


  // 0 if a borrow is required

  cpu_updateCarry_subtract(cpu, -1, 0);
  assert(cpu_getCarry(cpu) == TRUE);

  cpu_updateCarry_subtract(cpu, 1, 2);
  assert(cpu_getCarry(cpu) == FALSE);

  cpu_updateCarry_subtract(cpu, -100, 50);
  assert(cpu_getCarry(cpu) == TRUE);

  cpu_updateCarry_subtract(cpu, 100, 0);
  assert(cpu_getCarry(cpu) == TRUE);

  cpu_updateCarry_subtract(cpu, 100, 99);
  assert(cpu_getCarry(cpu) == TRUE);

  cpu_updateCarry_subtract(cpu, -1, -2);
  assert(cpu_getCarry(cpu) == TRUE);

  cpu_updateCarry_subtract(cpu, -2, -1);
  assert(cpu_getCarry(cpu) == FALSE);


}

{

  CPU cpu = cpu_init();
  assert(cpu != NULL);

  cpu_setOverflow(cpu, TRUE);
  assert(cpu_getOverflow(cpu) == TRUE);
  assert(cpu_getCarry(cpu) == FALSE);

  cpu_setOverflow(cpu, FALSE);
  assert(cpu_getOverflow(cpu) == FALSE);
  assert(cpu_getCarry(cpu) == FALSE);

  cpu_updateOverflow(cpu, -1, -2, 10);
  assert(cpu_getOverflow(cpu) == TRUE);

  cpu_updateOverflow(cpu, -1, -2, -3);
  assert(cpu_getOverflow(cpu) == FALSE);

  cpu_updateOverflow(cpu, 1, 2, 3);
  assert(cpu_getOverflow(cpu) == FALSE);

  cpu_updateOverflow(cpu, 1, 2, -3);
  assert(cpu_getOverflow(cpu) == TRUE);

  cpu_updateOverflow(cpu, -1, 2, -3);
  assert(cpu_getOverflow(cpu) == FALSE);

  cpu_updateOverflow(cpu, 1, -2, -3);
  assert(cpu_getOverflow(cpu) == FALSE);


  cpu_updateOverflow(cpu, -1, 2, 3);
  assert(cpu_getOverflow(cpu) == FALSE);
```

```
  cpu_updateOverflow(cpu, 1, -2, 3);
  assert(cpu_getOverflow(cpu) == FALSE);


  cpu_updateOverflow(cpu, 1, 1, 2);
  assert(cpu_getOverflow(cpu) == FALSE);

  cpu_updateOverflow(cpu, 1, -1, 0);
  assert(cpu_getOverflow(cpu) == FALSE);

  cpu_updateOverflow(cpu, 127, 1, 128);
  assert(cpu_getOverflow(cpu) == TRUE);

  cpu_updateOverflow(cpu, -128, -1, 127);
  assert(cpu_getOverflow(cpu) == TRUE);

  cpu_updateOverflow(cpu, 0, -1, -1);
  assert(cpu_getOverflow(cpu) == FALSE);

  cpu_updateOverflow(cpu, 127, 1, 128);
  assert(cpu_getOverflow(cpu) == TRUE);

}

{

  CPU cpu = cpu_init();
  assert(cpu != NULL);

  cpu_setInterruptDisable(cpu, TRUE);
  assert(cpu_getInterruptDisable(cpu) == TRUE);
  assert(cpu_getOverflow(cpu) == FALSE);
  assert(cpu_getCarry(cpu) == FALSE);
  assert(cpu_getNegative(cpu) == FALSE);
  assert(cpu_getZero(cpu) == FALSE);

  cpu_setInterruptDisable(cpu, FALSE);
  assert(cpu_getInterruptDisable(cpu) == FALSE);
  assert(cpu_getOverflow(cpu) == FALSE);
  assert(cpu_getCarry(cpu) == FALSE);
  assert(cpu_getNegative(cpu) == FALSE);
  assert(cpu_getZero(cpu) == FALSE);
}

{
  CPU cpu = cpu_init();
  assert(cpu != NULL);

  cpu_setBreak(cpu, TRUE);
  assert(cpu_getBreak(cpu) == TRUE);
  assert(cpu_getInterruptDisable(cpu) == FALSE);
  assert(cpu_getOverflow(cpu) == FALSE);
  assert(cpu_getCarry(cpu) == FALSE);
  assert(cpu_getNegative(cpu) == FALSE);
  assert(cpu_getZero(cpu) == FALSE);
  assert(cpu_getDecimal(cpu) == FALSE);

  cpu_setBreak(cpu, FALSE);
  assert(cpu_getBreak(cpu) == FALSE);
  assert(cpu_getInterruptDisable(cpu) == FALSE);
  assert(cpu_getOverflow(cpu) == FALSE);
  assert(cpu_getCarry(cpu) == FALSE);
  assert(cpu_getNegative(cpu) == FALSE);
  assert(cpu_getZero(cpu) == FALSE);
  assert(cpu_getDecimal(cpu) == FALSE);

  cpu_setInterruptDisable(cpu, TRUE);
  assert(cpu_getBreak(cpu) == FALSE);
  assert(cpu_getInterruptDisable(cpu) == TRUE);
  assert(cpu_getOverflow(cpu) == FALSE);
  assert(cpu_getCarry(cpu) == FALSE);
  assert(cpu_getNegative(cpu) == FALSE);
```

```
assert(cpu_getZero(cpu) == FALSE);
assert(cpu_getDecimal(cpu) == FALSE);

cpu_setInterruptDisable(cpu, FALSE);
assert(cpu_getBreak(cpu) == FALSE);
assert(cpu_getInterruptDisable(cpu) == FALSE);
assert(cpu_getOverflow(cpu) == FALSE);
assert(cpu_getCarry(cpu) == FALSE);
assert(cpu_getNegative(cpu) == FALSE);
assert(cpu_getZero(cpu) == FALSE);
assert(cpu_getDecimal(cpu) == FALSE);

cpu_setOverflow(cpu, TRUE);
assert(cpu_getBreak(cpu) == FALSE);
assert(cpu_getInterruptDisable(cpu) == FALSE);
assert(cpu_getOverflow(cpu) == TRUE);
assert(cpu_getCarry(cpu) == FALSE);
assert(cpu_getNegative(cpu) == FALSE);
assert(cpu_getZero(cpu) == FALSE);
assert(cpu_getDecimal(cpu) == FALSE);

cpu_setOverflow(cpu, FALSE);
assert(cpu_getBreak(cpu) == FALSE);
assert(cpu_getInterruptDisable(cpu) == FALSE);
assert(cpu_getOverflow(cpu) == FALSE);
assert(cpu_getCarry(cpu) == FALSE);
assert(cpu_getNegative(cpu) == FALSE);
assert(cpu_getZero(cpu) == FALSE);
assert(cpu_getDecimal(cpu) == FALSE);

cpu_setCarry(cpu, TRUE);
assert(cpu_getBreak(cpu) == FALSE);
assert(cpu_getInterruptDisable(cpu) == FALSE);
assert(cpu_getOverflow(cpu) == FALSE);
assert(cpu_getCarry(cpu) == TRUE);
assert(cpu_getNegative(cpu) == FALSE);
assert(cpu_getZero(cpu) == FALSE);
assert(cpu_getDecimal(cpu) == FALSE);

cpu_setCarry(cpu, FALSE);
assert(cpu_getBreak(cpu) == FALSE);
assert(cpu_getInterruptDisable(cpu) == FALSE);
assert(cpu_getOverflow(cpu) == FALSE);
assert(cpu_getCarry(cpu) == FALSE);
assert(cpu_getNegative(cpu) == FALSE);
assert(cpu_getZero(cpu) == FALSE);
assert(cpu_getDecimal(cpu) == FALSE);

cpu_setNegative(cpu, TRUE);
assert(cpu_getBreak(cpu) == FALSE);
assert(cpu_getInterruptDisable(cpu) == FALSE);
assert(cpu_getOverflow(cpu) == FALSE);
assert(cpu_getCarry(cpu) == FALSE);
assert(cpu_getNegative(cpu) == TRUE);
assert(cpu_getZero(cpu) == FALSE);
assert(cpu_getDecimal(cpu) == FALSE);

cpu_setNegative(cpu, FALSE);
assert(cpu_getBreak(cpu) == FALSE);
assert(cpu_getInterruptDisable(cpu) == FALSE);
assert(cpu_getOverflow(cpu) == FALSE);
assert(cpu_getCarry(cpu) == FALSE);
assert(cpu_getNegative(cpu) == FALSE);
assert(cpu_getZero(cpu) == FALSE);
assert(cpu_getDecimal(cpu) == FALSE);

cpu_setZero(cpu, TRUE);
assert(cpu_getBreak(cpu) == FALSE);
assert(cpu_getInterruptDisable(cpu) == FALSE);
assert(cpu_getOverflow(cpu) == FALSE);
assert(cpu_getCarry(cpu) == FALSE);
assert(cpu_getNegative(cpu) == FALSE);
```

```
    assert(cpu_getZero(cpu) == TRUE);
    assert(cpu_getDecimal(cpu) == FALSE);

    cpu_setZero(cpu, FALSE);
    assert(cpu_getBreak(cpu) == FALSE);
    assert(cpu_getInterruptDisable(cpu) == FALSE);
    assert(cpu_getOverflow(cpu) == FALSE);
    assert(cpu_getCarry(cpu) == FALSE);
    assert(cpu_getNegative(cpu) == FALSE);
    assert(cpu_getZero(cpu) == FALSE);
    assert(cpu_getDecimal(cpu) == FALSE);

    cpu_setDecimal(cpu, TRUE);
    assert(cpu_getBreak(cpu) == FALSE);
    assert(cpu_getInterruptDisable(cpu) == FALSE);
    assert(cpu_getOverflow(cpu) == FALSE);
    assert(cpu_getCarry(cpu) == FALSE);
    assert(cpu_getNegative(cpu) == FALSE);
    assert(cpu_getZero(cpu) == FALSE);
    assert(cpu_getDecimal(cpu) == TRUE);

    cpu_setDecimal(cpu, FALSE);
    assert(cpu_getBreak(cpu) == FALSE);
    assert(cpu_getInterruptDisable(cpu) == FALSE);
    assert(cpu_getOverflow(cpu) == FALSE);
    assert(cpu_getCarry(cpu) == FALSE);
    assert(cpu_getNegative(cpu) == FALSE);
    assert(cpu_getZero(cpu) == FALSE);
    assert(cpu_getDecimal(cpu) == FALSE);
  }

}
```

cpu.h

```
#include <assert.h>
#include "cpu_type.h"
#include "globals.h"
#include "nes_type.h"

#define MASK_STATUS_ZERO_ON MASK_BIT1
#define MASK_STATUS_ZERO_OFF (~MASK_BIT1)

#define MASK_STATUS_DECIMAL_ON MASK_BIT3
#define MASK_STATUS_DECIMAL_OFF (~MASK_BIT3)

#define MASK_STATUS_NEGATIVE_ON MASK_BIT7
#define MASK_STATUS_NEGATIVE_OFF (~MASK_BIT7)

#define MASK_STATUS_OVERFLOW_ON  MASK_BIT6
#define MASK_STATUS_OVERFLOW_OFF (~MASK_BIT6)

#define MASK_STATUS_CARRY_ON MASK_BIT0
#define MASK_STATUS_CARRY_OFF (~MASK_BIT0)

#define MASK_STATUS_INTERRUPT_ON MASK_BIT2
#define MASK_STATUS_INTERRUPT_OFF (~MASK_BIT2)

#define MASK_STATUS_BREAK_ON MASK_BIT4
#define MASK_STATUS_BREAK_OFF (~MASK_BIT4)

#define CPU_STATUS_REGISTER_INITIAL_VALUE MASK_BIT5

#define CPU_STACK_POINTER_INITIAL_VALUE 0xFF

#define GET_STACK_ADDRESS(X) ((Address) 0x0100 + X)

#define STACK_ADDRESS_TOP 0x01FF
#define STACK_ADDRESS_BOTTOM 0x0100

#define VALIDATE_STACK_ADDRESS(X) {assert(X <= STACK_ADDRESS_TOP); assert(X >=
STACK_ADDRESS_BOTTOM);}

#define VALIDATE_STACK_POINTER(X) VALIDATE_STACK_ADDRESS(GET_STACK_ADDRESS(X))


CPU cpu_init(void);

void cpu_destroy(CPU cpu);

Address cpu_getProgramCounter(CPU cpu);
Byte cpu_getStackPointer(CPU cpu);
Byte cpu_getStatus(CPU cpu);
Byte cpu_getIndexX(CPU cpu);
Byte cpu_getIndexY(CPU cpu);
Byte cpu_getAccumulator(CPU cpu);

void cpu_setProgramCounter(CPU cpu, Address programCounter);
void cpu_setStackPointer(CPU cpu, Byte stackPointer);
void cpu_setStatus(CPU cpu, Byte status);
void cpu_setIndexX(CPU cpu, Byte indexX);
void cpu_setIndexY(CPU cpu, Byte indexY);
void cpu_setAccumulator(CPU cpu, Byte accumulator);

void cpu_step(NES nes);

void cpu_tests(void);

void cpu_handleInterrupt(NES nes, Address handlerLowByte, Boolean fromBRK);

Boolean cpu_getInterruptDisable(CPU cpu);
```

cpu_type.h

```
#ifndef CPU_TYPE_H
  #define CPU_TYPE_H
  typedef struct cpu *CPU;
#endif
```

cpuMemory.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "globals.h"
#include "memory.h"
#include "cpuMemory.h"
#include "nes.h"
#include "ppu.h"
#include "ppuMemory.h"
#include "objectAttributeMemory.h"

//////////////////////////////////////////////////////////////////////////////////
//

static Address cpuMemory_ramMirror_getLowestAddress(Address address) {

  while(address > CPU_GENUINE_RAM_LAST_ADDRESS) {
    address -= CPU_RAM_MIRRORED_SIZE;
  }

  assert(address <= CPU_GENUINE_RAM_LAST_ADDRESS);

  return address;
}

static void cpuMemory_ramMirror_writer(NES nes, Address address, Byte data) {
  // Memory locations $0000-$07FF are mirrored three times at $0800-$1FFF.
  // This means that, for example, any data written to $0000 will also be written to
$0800, $1000 and $1800.
  address = cpuMemory_ramMirror_getLowestAddress(address);

  Memory cpuMemory = nes_getCPUMemory(nes);
  assert(cpuMemory != NULL);

  memory_write_callback(nes, cpuMemory, address, data);
}

static Byte cpuMemory_ramMirror_reader(NES nes, Address address) {
  // Memory locations $0000-$07FF are mirrored three times at $0800-$1FFF.
  // This means that, for example, any data written to $0000 will also be written to
$0800, $1000 and $1800.
  address = cpuMemory_ramMirror_getLowestAddress(address);

  Memory cpuMemory = nes_getCPUMemory(nes);
  assert(cpuMemory != NULL);

  Byte data = memory_read_callback(nes, cpuMemory, address);

  return data;
}

//////////////////////////////////////////////////////////////////////////////////
//

static Address cpuMemory_ppuMirror_getLowestAddress(Address address) {

  while(address > CPU_GENUINE_PPU_LAST_ADDRESS) {
    address -= CPU_PPU_MIRRORED_SIZE;
  }

  assert(address <= CPU_GENUINE_PPU_LAST_ADDRESS);

  return address;
}

static void cpuMemory_ppuMirror_writer(NES nes, Address address, Byte data) {
  address = cpuMemory_ppuMirror_getLowestAddress(address);

  Memory cpuMemory = nes_getCPUMemory(nes);
```

```
  assert(cpuMemory != NULL);

  memory_write_callback(nes, cpuMemory, address, data);
}

static Byte cpuMemory_ppuMirror_reader(NES nes, Address address) {
  address = cpuMemory_ppuMirror_getLowestAddress(address);

  Memory cpuMemory = nes_getCPUMemory(nes);
  assert(cpuMemory != NULL);

  Byte data = memory_read_callback(nes, cpuMemory, address);

  return data;
}

////////////////////////////////////////////////////////////////////////////////
///

// #define CPU_PPU_CONTROL_REGISTER_ADDRESS                0x2000 // write

static Byte cpuMemory_ppuControlRegister_reader(NES nes, Address address) {
  assert(nes != NULL);

  // not allowed to read this
  // assert(FALSE);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  return ppu_getControlRegister(ppu);
}

static void cpuMemory_ppuControlRegister_writer(NES nes, Address address, Byte data) {
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  ppu_setControlRegister(ppu, data);
}

////////////////////////////////////////////////////////////////////////////////
///

// #define CPU_PPU_MASK_REGISTER_ADDRESS                   0x2001 // write

static Byte cpuMemory_ppuMaskRegister_reader(NES nes, Address address) {
  assert(nes != NULL);

  // not allowed to read this
  // assert(FALSE);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  return ppu_getMaskRegister(ppu);
}

static void cpuMemory_ppuMaskRegister_writer(NES nes, Address address, Byte data) {
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  ppu_setMaskRegister(ppu, data);
}

////////////////////////////////////////////////////////////////////////////////
///

// #define CPU_PPU_STATUS_REGISTER_ADDRESS                 0x2002 // read
```

```
static Byte cpuMemory_ppuStatusRegister_reader(NES nes, Address address) {
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  return ppu_getStatusRegister(ppu);
}

static void cpuMemory_ppuStatusRegister_writer(NES nes, Address address, Byte data) {
  assert(nes != NULL);

  // not allowed to write this
  assert(FALSE);
}

////////////////////////////////////////////////////////////////////////////////
///

// #define CPU_PPU_SPRITE_ADDRESS_REGISTER_ADDRESS         0x2003 // write

static Byte cpuMemory_ppuSpriteAddressRegister_reader(NES nes, Address address) {
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  return ppu_getSpriteAddressRegister(ppu);
}

static void cpuMemory_ppuSpriteAddressRegister_writer(NES nes, Address address, Byte
data) {
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  ppu_setSpriteAddressRegister(ppu, data);
}


////////////////////////////////////////////////////////////////////////////////
///

// #define CPU_PPU_SPRITE_DATA_REGISTER_ADDRESS            0x2004 // write

static Byte cpuMemory_ppuSpriteDataRegister_reader(NES nes, Address address) {
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  return ppu_getSpriteDataRegister(nes);
}

static void cpuMemory_ppuSpriteDataRegister_writer(NES nes, Address address, Byte data)
{
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  ppu_setSpriteDataRegister(nes, data);
}

////////////////////////////////////////////////////////////////////////////////
///

// #define CPU_PPU_SCROLL_REGISTER_ADDRESS                 0x2005 // write

static Byte cpuMemory_ppuScrollRegister_reader(NES nes, Address address) {
```

```
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  return ppu_getScrollRegister(ppu);
}

static void cpuMemory_ppuScrollRegister_writer(NES nes, Address address, Byte data) {
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  ppu_setScrollRegister(ppu, data);
}

////////////////////////////////////////////////////////////////////////////////
///

// #define CPU_PPUMEMORY_ADDRESS_REGISTER_ADDRESS          0x2006 // write

static Byte cpuMemory_ppuMemoryAddressRegister_reader(NES nes, Address address) {
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  return ppu_getPPUMemoryAddressRegister(ppu);
}

static void cpuMemory_ppuMemoryAddressRegister_writer(NES nes, Address address, Byte
data) {
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  ppu_setPPUMemoryAddressRegister(ppu, data);
}



////////////////////////////////////////////////////////////////////////////////
///

// #define CPU_PPUMEMORY_DATA_REGISTER_ADDRESS             0x2007 // read/write

static Byte cpuMemory_ppuMemoryDataRegister_reader(NES nes, Address address) {
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  return ppu_getPPUMemoryDataRegister(nes);
}

static void cpuMemory_ppuMemoryDataRegister_writer(NES nes, Address address, Byte data)
{
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  ppu_setPPUMemoryDataRegister(nes, data);
}


////////////////////////////////////////////////////////////////////////////////
///

// #define CPU_SPRITE_DMA_REGISTER_ADDRESS                 0x4014 // write
```

```
#define CPU_DMA_ADDRESS_MULTIPLIER 0x0100

static Byte cpuMemory_spriteDMARegister_reader(NES nes, Address address) {
  assert(nes != NULL);

  return 0;
}

static void cpuMemory_spriteDMARegister_writer(NES nes, Address address, Byte data) {
  assert(nes != NULL);

  Address readAddress = data * CPU_DMA_ADDRESS_MULTIPLIER;

  Address offset;

  Memory cpuMemory = nes_getCPUMemory(nes);
  assert(cpuMemory != NULL);

  Memory objectAttributeMemory = nes_getObjectAttributeMemory(nes);
  assert(objectAttributeMemory != NULL);

  for (offset=0; offset < OAM_NUM_ADDRESSES; offset++) {

    Byte data = memory_read_callback(nes, cpuMemory, readAddress + offset);

    nes_cpuCycled(nes);

    memory_write_callback(nes, objectAttributeMemory, offset, data);

    nes_cpuCycled(nes);
  }
}

////////////////////////////////////////////////////////////////////////////////////////
///

// CPU_JOYPAD_0_ADDRESS
// CPU_JOYPAD_1_ADDRESS

static Byte cpuMemory_joypad_reader(NES nes, Address address) {
  assert(nes != NULL);

  int currentJoypad = address - CPU_JOYPAD_0_ADDRESS;
  assert(currentJoypad >= 0);
  assert(currentJoypad <= 3);

  return nes_readJoypad(nes, currentJoypad);
}

static void cpuMemory_joypad_writer(NES nes, Address address, Byte data) {
  assert(nes != NULL);

  int currentJoypad = address - CPU_JOYPAD_0_ADDRESS;
  assert(currentJoypad >= 0);
  assert(currentJoypad <= 3);

  nes_writeJoypad(nes, currentJoypad, data);
}

////////////////////////////////////////////////////////////////////////////////////////
///

Memory cpuMemory_init(void) {
  Memory memory = memory_init(CPU_TOTAL_MEMORY_ADDRESSES);
  assert(memory != NULL);

  int address;


  //for (address=CPU_FIRST_RAM_MIRRORED_ADDRESS; address <=
CPU_LAST_RAM_MIRRORED_ADDRESS; address++) {
```

```
  for (address=CPU_RAM_MIRROR_FIRST_ADDRESS; address <= CPU_RAM_MIRROR_LAST_ADDRESS;
address++) {
    memory_setWriteCallback(memory, address, &cpuMemory_ramMirror_writer);
    memory_setReadCallback(memory, address, &cpuMemory_ramMirror_reader);
  }

  // setup the ppu register redirectors

  memory_setReadCallback(memory, CPU_PPU_CONTROL_REGISTER_ADDRESS,
&cpuMemory_ppuControlRegister_reader);
  memory_setWriteCallback(memory, CPU_PPU_CONTROL_REGISTER_ADDRESS,
&cpuMemory_ppuControlRegister_writer);

  memory_setReadCallback(memory, CPU_PPU_MASK_REGISTER_ADDRESS,
&cpuMemory_ppuMaskRegister_reader);
  memory_setWriteCallback(memory, CPU_PPU_MASK_REGISTER_ADDRESS,
&cpuMemory_ppuMaskRegister_writer);

  memory_setReadCallback(memory, CPU_PPU_STATUS_REGISTER_ADDRESS,
&cpuMemory_ppuStatusRegister_reader);
  memory_setWriteCallback(memory, CPU_PPU_STATUS_REGISTER_ADDRESS,
&cpuMemory_ppuStatusRegister_writer);

  memory_setReadCallback(memory, CPU_PPU_SPRITE_ADDRESS_REGISTER_ADDRESS,
&cpuMemory_ppuSpriteAddressRegister_reader);
  memory_setWriteCallback(memory, CPU_PPU_SPRITE_ADDRESS_REGISTER_ADDRESS,
&cpuMemory_ppuSpriteAddressRegister_writer);

  memory_setReadCallback(memory, CPU_PPU_SPRITE_DATA_REGISTER_ADDRESS,
&cpuMemory_ppuSpriteDataRegister_reader);
  memory_setWriteCallback(memory, CPU_PPU_SPRITE_DATA_REGISTER_ADDRESS,
&cpuMemory_ppuSpriteDataRegister_writer);

  memory_setReadCallback(memory, CPU_PPU_SCROLL_REGISTER_ADDRESS,
&cpuMemory_ppuScrollRegister_reader);
  memory_setWriteCallback(memory, CPU_PPU_SCROLL_REGISTER_ADDRESS,
&cpuMemory_ppuScrollRegister_writer);

  memory_setReadCallback(memory, CPU_PPUMEMORY_ADDRESS_REGISTER_ADDRESS,
&cpuMemory_ppuMemoryAddressRegister_reader);
  memory_setWriteCallback(memory, CPU_PPUMEMORY_ADDRESS_REGISTER_ADDRESS,
&cpuMemory_ppuMemoryAddressRegister_writer);

  memory_setReadCallback(memory, CPU_PPUMEMORY_DATA_REGISTER_ADDRESS,
&cpuMemory_ppuMemoryDataRegister_reader);
  memory_setWriteCallback(memory, CPU_PPUMEMORY_DATA_REGISTER_ADDRESS,
&cpuMemory_ppuMemoryDataRegister_writer);

  // now setup their mirrors

  for (address=CPU_PPU_MIRROR_FIRST_ADDRESS; address <= CPU_PPU_MIRROR_LAST_ADDRESS;
address++) {
    memory_setWriteCallback(memory, address, &cpuMemory_ppuMirror_writer);
    memory_setReadCallback(memory, address, &cpuMemory_ppuMirror_reader);
  }

  memory_setReadCallback(memory, CPU_SPRITE_DMA_REGISTER_ADDRESS,
&cpuMemory_spriteDMARegister_reader);
  memory_setWriteCallback(memory, CPU_SPRITE_DMA_REGISTER_ADDRESS,
&cpuMemory_spriteDMARegister_writer);


  memory_setReadCallback(memory, CPU_JOYPAD_0_ADDRESS, &cpuMemory_joypad_reader);
  memory_setWriteCallback(memory, CPU_JOYPAD_0_ADDRESS, &cpuMemory_joypad_writer);

  memory_setReadCallback(memory, CPU_JOYPAD_1_ADDRESS, &cpuMemory_joypad_reader);
  memory_setWriteCallback(memory, CPU_JOYPAD_1_ADDRESS, &cpuMemory_joypad_writer);

  return memory;
}
```

cpuMemory.h

```
#include "memory_type.h"

Memory cpuMemory_init(void);


#define CPU_MEMORY_FIRST_ADDRESS 0x0000
#define CPU_MEMORY_LAST_ADDRESS 0xFFFF

#define CPU_TOTAL_MEMORY_ADDRESSES (CPU_MEMORY_LAST_ADDRESS + 1)


#define CPU_RESET_VECTOR_LOWER_ADDRESS 0xFFFC
#define CPU_RESET_VECTOR_UPPER_ADDRESS (CPU_RESET_VECTOR_LOWER_ADDRESS + 1)

#define CPU_IRQ_VECTOR_LOWER_ADDRESS 0xFFFE
#define CPU_IRQ_VECTOR_UPPER_ADDRESS (CPU_IRQ_VECTOR_LOWER_ADDRESS + 1)

#define CPU_NMI_VECTOR_LOWER_ADDRESS 0xFFFA
#define CPU_NMI_VECTOR_UPPER_ADDRESS (CPU_NMI_VECTOR_LOWER_ADDRESS + 1)


#define CPU_PROGRAM_LOWER_FIRST_ADDRESS 0x8000
#define CPU_PROGRAM_LOWER_LAST_ADDRESS 0xBFFF

#define CPU_PROGRAM_UPPER_FIRST_ADDRESS 0xC000
#define CPU_PROGRAM_UPPER_LAST_ADDRESS  0xFFFF



#define CPU_GENUINE_RAM_FIRST_ADDRESS 0x000
#define CPU_GENUINE_RAM_LAST_ADDRESS 0x07FF

#define CPU_RAM_MIRRORED_SIZE (CPU_GENUINE_RAM_LAST_ADDRESS -
CPU_GENUINE_RAM_FIRST_ADDRESS + 1)

#define CPU_RAM_MIRROR_FIRST_ADDRESS (CPU_GENUINE_RAM_LAST_ADDRESS + 1)

#define CPU_RAM_MIRROR_LAST_ADDRESS 0x1FFF

#define CPU_RAM_MIRROR_SIZE (CPU_RAM_MIRROR_LAST_ADDRESS - CPU_RAM_MIRROR_FIRST_ADDRESS
+ 1)

#define CPU_NUM_RAM_MIRRORS (CPU_RAM_MIRROR_SIZE / CPU_RAM_MIRRORED_SIZE)




#define CPU_GENUINE_PPU_FIRST_ADDRESS 0x2000
#define CPU_GENUINE_PPU_LAST_ADDRESS 0x2007

#define CPU_PPU_MIRRORED_SIZE (CPU_GENUINE_PPU_LAST_ADDRESS -
CPU_GENUINE_PPU_FIRST_ADDRESS +1)

#define CPU_PPU_MIRROR_FIRST_ADDRESS (CPU_GENUINE_PPU_LAST_ADDRESS + 1)

#define CPU_PPU_MIRROR_LAST_ADDRESS 0x3FFF



#define CPU_PPU_MIRROR_SIZE (CPU_PPU_MIRROR_LAST_ADDRESS - CPU_PPU_MIRROR_FIRST_ADDRESS
+ 1)

#define CPU_NUM_PPU_MIRRORS (CPU_PPU_MIRROR_SIZE / CPU_PPU_MIRRORED_SIZE)



#define CPU_PPU_CONTROL_REGISTER_ADDRESS               0x2000 // write
#define CPU_PPU_MASK_REGISTER_ADDRESS                  0x2001 // write
#define CPU_PPU_STATUS_REGISTER_ADDRESS                0x2002 // read
```

```
// internal object attribute memory index pointer (64 attributes, 32 bits
// each, byte granular access). stored value post-increments on access to port 4.

#define CPU_PPU_SPRITE_ADDRESS_REGISTER_ADDRESS         0x2003 // write

// returns object attribute memory location indexed by port 3, then increments port 3.

#define CPU_PPU_SPRITE_DATA_REGISTER_ADDRESS            0x2004 // write

// 5    -       scroll offset port.

#define CPU_PPU_SCROLL_REGISTER_ADDRESS                 0x2005 // write

// 6    -       PPU address port to access with port 7.

#define CPU_PPUMEMORY_ADDRESS_REGISTER_ADDRESS          0x2006 // write

// 7    -       PPU memory write port.

#define CPU_PPUMEMORY_DATA_REGISTER_ADDRESS             0x2007 // read/write



#define CPU_APU_PULSE_1_CONTROL_REGISTER_ADDRESS        0x4000 // write
#define CPU_APU_PULSE_1_RAMP_CONTROL_REGISTER_ADDRESS   0x4001 // write
#define CPU_APU_PULSE_1_FINE_TUNE_REGISTER_ADDRESS      0x4002 // write
#define CPU_APU_PULSE_1_COARSE_TUNE_REGISTER_ADDRESS    0x4003 // write

#define CPU_APU_PULSE_2_CONTROL_REGISTER_ADDRESS        0x4004 // write
#define CPU_APU_PULSE_2_RAMP_CONTROL_REGISTER_ADDRESS   0x4005 // write
#define CPU_APU_PULSE_2_FINE_TUNE_REGISTER_ADDRESS      0x4006 // write
#define CPU_APU_PULSE_2_COARSE_TUNE_REGISTER_ADDRESS    0x4007 // write

#define CPU_APU_TRIANGLE_CONTROL_REGISTER_1_ADDRESS     0x4008 // write
#define CPU_APU_TRIANGLE_CONTROL_REGISTER_2_ADDRESS     0x4009 // write

#define CPU_APU_TRIANGLE_FREQUENCY_REGISTER_1_ADDRESS   0x400A // write
#define CPU_APU_TRIANGLE_FREQUENCY_REGISTER_2_ADDRESS   0x400B // write

#define CPU_APU_NOISE_CONTROL_REGISTER_1_ADDRESS        0x400C // write
                                                //      0x400D ??
#define CPU_APU_NOISE_FREQUENCY_REGISTER_1_ADDRESS      0x400E // write
#define CPU_APU_NOISE_FREQUENCY_REGISTER_2_ADDRESS      0x400F // write

#define CPU_APU_DELTA_CONTROL_REGISTER_ADDRESS          0x4010 // write
#define CPU_APU_DELTA_DA_REGISTER_ADDRESS               0x4011 // write
#define CPU_APU_DELTA_ADDRESS_REGISTER_ADDRESS          0x4012 // write
#define CPU_APU_DELTA_DATE_LENGTH_REGISTER_ADDRESS      0x4013 // write


#define CPU_SPRITE_DMA_REGISTER_ADDRESS                 0x4014 // write

#define CPU_APU_VERTICAL_CLOCK_REGISTER_ADDRESS         0x4015 // read/write

#define CPU_JOYPAD_0_ADDRESS                            0x4016 // read/write
#define CPU_JOYPAD_1_ADDRESS                            0x4017 // read/write
```

emulator.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "nes.h"

#define DEFAULT_RESOLUTION_WIDTH 256
#define DEFAULT_RESOLUTION_HEIGHT 240

int main(int argc, char *argv[]) {

  assert(argc >= 2);

  char *filename = NULL;

  int width = DEFAULT_RESOLUTION_WIDTH;
  int height = DEFAULT_RESOLUTION_HEIGHT;

  if (argc == 2) {

    filename = argv[1];

  } else if (argc == 4) {

    width = atoi(argv[1]);
    height = atoi(argv[2]);
    filename = argv[3];

  }

  NES nes = nes_init(filename, width, height);

  nes_run(nes);

  nes_destroy(nes);

  return 0;
}
```

globals.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <stdarg.h>
#include "globals.h"

#define LOG_LINE_LENGTH 1024

void debug_printf (const char *format, ... ) {

#ifdef LOG
  va_list arg;
  va_start (arg, format);

  char logBuffer[LOG_LINE_LENGTH];
  vsnprintf(logBuffer,LOG_LINE_LENGTH-1,format,arg);

  fprintf(stderr,logBuffer,LOG_LINE_LENGTH);

  va_end (arg);
#endif

}
```

globals.h

```
#ifndef GLOBALS_H

  #define GLOBALS_H

  #include <stdint.h>

  #define MASK_BIT0 1
  #define MASK_BIT1 2
  #define MASK_BIT2 4
  #define MASK_BIT3 8
  #define MASK_BIT4 16
  #define MASK_BIT5 32
  #define MASK_BIT6 64
  #define MASK_BIT7 128

  #define MASK_BIT8 256

  #define BITS_PER_BYTE 8

  #define BYTES_PER_KILOBYTE 1024

  typedef enum {
    FALSE = 0,
    TRUE = 1
  } Boolean;

  typedef uint8_t Byte;

  typedef int8_t SignedByte;

  typedef uint16_t Word;

  typedef int16_t SignedWord;

  typedef uint16_t Address;

  #define GET_ADDRESS_HIGH_BYTE(X) ((Byte) (X >> BITS_PER_BYTE))

  #define GET_ADDRESS_LOW_BYTE(X) ((Byte) X)

  void debug_printf (const char *format, ... );

#endif
```

gui.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "globals.h"
#include "gui.h"
#include <SDL/SDL.h>
#include <string.h>
#include "joypad.h"

#define GUI_MAX_JOYPADS 4
#define GUI_IMAGE_SCALE 2

struct gui {
  SDL_Surface *screen;
  int width;
  int height;
  int currentFrame;
  Boolean buttonState[GUI_MAX_JOYPADS][JOYPAD_NUM_BUTTONS];
  Boolean receivedTerminationRequest;
};

int gui_getWidth(GUI gui) {
  assert(gui != NULL);
  return gui->width;
}

int gui_getHeight(GUI gui) {
  assert(gui != NULL);
  return gui->height;
}

void gui_drawPixel(GUI gui, Byte x, Byte y, Byte red, Byte green, Byte blue) {
  assert(gui != NULL);
  assert(gui->screen != NULL);

  assert(x < gui->width);    // Byte type is unsigned
  assert(y < gui->height);   // Byte type is unsigned

  Uint32 color = SDL_MapRGB(gui->screen->format, red, green, blue);

  assert(gui->screen->format->BytesPerPixel == 4);

  Uint32 *bufp;

  // hack, hardcoded scale of 2

  Word actualX = GUI_IMAGE_SCALE * x;
  Word actualY = GUI_IMAGE_SCALE * y;

  bufp = (Uint32 *)gui->screen->pixels + ((actualY)*gui->screen->pitch/4) + (actualX);
  *bufp = color;

  bufp = (Uint32 *)gui->screen->pixels + ((actualY)*gui->screen->pitch/4) + (actualX) +
1;
  *bufp = color;

  bufp = (Uint32 *)gui->screen->pixels + (((actualY)+1)*gui->screen->pitch/4) +
(actualX);
  *bufp = color;

  bufp = (Uint32 *)gui->screen->pixels + (((actualY)+1)*gui->screen->pitch/4) +
(actualX) + 1;
  *bufp = color;
}

void gui_refresh(GUI gui) {
  assert(gui != NULL);
  assert(gui->screen != NULL);
```

```
  if (gui->currentFrame % 200 == 0) {
    char filename[100];
    sprintf(filename, "capture%03d.bmp", gui->currentFrame);
    SDL_SaveBMP(gui->screen,filename);
  }

  gui->currentFrame++;

  SDL_Flip(gui->screen);

        //SDL_Delay(1000/60);
}

static void gui_resetButtonState(GUI gui) {
  assert(gui != NULL);

  int joypad;
  for (joypad=0; joypad < GUI_MAX_JOYPADS; joypad++) {
    int button;
    for (button=0; button < JOYPAD_NUM_BUTTONS; button++) {
      gui->buttonState[joypad][button] = 0;
    }
  }
}

GUI gui_init(int width, int height) {
  GUI gui = (GUI) malloc(sizeof(struct gui));
  assert(gui != NULL);

                  //SDL_INIT_AUDIO
  assert(SDL_Init(SDL_INIT_VIDEO) >= 0);

  SDL_WM_SetCaption( "1337NES", 0 );

  atexit(SDL_Quit);

  gui->screen=SDL_SetVideoMode(width * GUI_IMAGE_SCALE,height *
GUI_IMAGE_SCALE,32,SDL_HWSURFACE|SDL_DOUBLEBUF); //|SDL_FULLSCREEN);

  assert(gui->screen != NULL);

  gui->width = width;
  gui->height = height;

  gui->currentFrame = 0;

  gui->receivedTerminationRequest = FALSE;

  gui_resetButtonState(gui);

  return gui;
}

Boolean gui_receivedTerminationRequest(GUI gui) {
  assert(gui != NULL);
  return gui->receivedTerminationRequest;
}

void gui_destroy(GUI gui) {
  assert(gui != NULL);
  free(gui);
}

void gui_queryInput(GUI gui) {
  assert(gui != NULL);

  //gui_resetButtonState(gui);

  SDL_Event event;

  while ( SDL_PollEvent(&event) ) {
```

```
if ( event.type == SDL_QUIT) {

  gui->receivedTerminationRequest = TRUE;

} else if ( event.type == SDL_KEYDOWN || event.type == SDL_KEYUP) {

  switch(event.key.keysym.sym) {

    case GUI_KEY_JOYPAD_0_BUTTON_A:
      gui->buttonState[0][JOYPAD_BUTTON_A] = (event.type == SDL_KEYDOWN);
      break;

    case GUI_KEY_JOYPAD_0_BUTTON_B:
      gui->buttonState[0][JOYPAD_BUTTON_B] = (event.type == SDL_KEYDOWN);
      break;

    case GUI_KEY_JOYPAD_0_BUTTON_START:
      gui->buttonState[0][JOYPAD_BUTTON_START] = (event.type == SDL_KEYDOWN);
      break;

    case GUI_KEY_JOYPAD_0_BUTTON_SELECT:
      gui->buttonState[0][JOYPAD_BUTTON_SELECT] = (event.type == SDL_KEYDOWN);
      break;

    case GUI_KEY_JOYPAD_0_BUTTON_UP:
      gui->buttonState[0][JOYPAD_BUTTON_UP] = (event.type == SDL_KEYDOWN);
      break;

    case GUI_KEY_JOYPAD_0_BUTTON_DOWN:
      gui->buttonState[0][JOYPAD_BUTTON_DOWN] = (event.type == SDL_KEYDOWN);
      break;

    case GUI_KEY_JOYPAD_0_BUTTON_LEFT:
      gui->buttonState[0][JOYPAD_BUTTON_LEFT] = (event.type == SDL_KEYDOWN);
      break;

    case GUI_KEY_JOYPAD_0_BUTTON_RIGHT:
      gui->buttonState[0][JOYPAD_BUTTON_RIGHT] = (event.type == SDL_KEYDOWN);
      break;



    case GUI_KEY_JOYPAD_1_BUTTON_A:
      gui->buttonState[1][JOYPAD_BUTTON_A] = (event.type == SDL_KEYDOWN);
      break;

    case GUI_KEY_JOYPAD_1_BUTTON_B:
      gui->buttonState[1][JOYPAD_BUTTON_B] = (event.type == SDL_KEYDOWN);
      break;

    case GUI_KEY_JOYPAD_1_BUTTON_START:
      gui->buttonState[1][JOYPAD_BUTTON_START] = (event.type == SDL_KEYDOWN);
      break;

    case GUI_KEY_JOYPAD_1_BUTTON_SELECT:
      gui->buttonState[1][JOYPAD_BUTTON_SELECT] = (event.type == SDL_KEYDOWN);
      break;

    case GUI_KEY_JOYPAD_1_BUTTON_UP:
      gui->buttonState[1][JOYPAD_BUTTON_UP] = (event.type == SDL_KEYDOWN);
      break;

    case GUI_KEY_JOYPAD_1_BUTTON_DOWN:
      gui->buttonState[1][JOYPAD_BUTTON_DOWN] = (event.type == SDL_KEYDOWN);
      break;

    case GUI_KEY_JOYPAD_1_BUTTON_LEFT:
      gui->buttonState[1][JOYPAD_BUTTON_LEFT] = (event.type == SDL_KEYDOWN);
      break;

    case GUI_KEY_JOYPAD_1_BUTTON_RIGHT:
```

```
          gui->buttonState[1][JOYPAD_BUTTON_RIGHT] = (event.type == SDL_KEYDOWN);
          break;

        case SDLK_ESCAPE:
          gui->receivedTerminationRequest = TRUE;
          break;

        default:
          printf("Unmapped key press: %d\n", event.key.keysym.sym);
          break;

      }
    }
  }
}

Boolean gui_isButtonPressed(GUI gui, int joypad, Button button) {
  assert(gui != NULL);

  assert(joypad >= 0);
  assert(joypad <= GUI_MAX_JOYPADS);

  assert(button >= 0);
  assert(button <= JOYPAD_NUM_BUTTONS);

  return gui->buttonState[joypad][button];
}
```

gui.h

```
#include "gui_type.h"

#include "globals.h"

#include "joypad_type.h"

#define GUI_KEY_JOYPAD_0_BUTTON_A       SDLK_q
#define GUI_KEY_JOYPAD_0_BUTTON_B       SDLK_e
#define GUI_KEY_JOYPAD_0_BUTTON_SELECT  SDLK_c
#define GUI_KEY_JOYPAD_0_BUTTON_START   SDLK_z
#define GUI_KEY_JOYPAD_0_BUTTON_UP      SDLK_w
#define GUI_KEY_JOYPAD_0_BUTTON_DOWN    SDLK_s
#define GUI_KEY_JOYPAD_0_BUTTON_LEFT    SDLK_a
#define GUI_KEY_JOYPAD_0_BUTTON_RIGHT   SDLK_d

// must turn NUM-LOCK on
#define GUI_KEY_JOYPAD_1_BUTTON_A       SDLK_KP7
#define GUI_KEY_JOYPAD_1_BUTTON_B       SDLK_KP9
#define GUI_KEY_JOYPAD_1_BUTTON_SELECT  SDLK_KP3
#define GUI_KEY_JOYPAD_1_BUTTON_START   SDLK_KP1
#define GUI_KEY_JOYPAD_1_BUTTON_UP      SDLK_KP8
#define GUI_KEY_JOYPAD_1_BUTTON_DOWN    SDLK_KP5
#define GUI_KEY_JOYPAD_1_BUTTON_LEFT    SDLK_KP4
#define GUI_KEY_JOYPAD_1_BUTTON_RIGHT   SDLK_KP6


void gui_destroy(GUI gui);

int gui_getWidth(GUI gui);

int gui_getHeight(GUI gui);

void gui_drawPixel(GUI gui, Byte x, Byte y, Byte red, Byte green, Byte blue);

void gui_refresh(GUI gui);

GUI gui_init(int width, int height);

void gui_queryInput(GUI gui);

Boolean gui_isButtonPressed(GUI gui, int joypad, Button button);


Boolean gui_receivedTerminationRequest(GUI gui);
```

gui_type.h

```
#ifndef GUI_TYPE_H
  #define GUI_TYPE_H
  typedef struct gui *GUI;
#endif
```

instructions.h

```
#define LDA_IMM 0xA9
#define LDX_IMM 0xA2
#define LDY_IMM 0xA0
#define AND_IMM 0x29
#define ORA_IMM 0x09
#define EOR_IMM 0x49
#define CMP_IMM 0xC9
#define CPX_IMM 0xE0
#define CPY_IMM 0xC0
#define ADC_IMM 0x69
#define SBC_IMM 0xE9

#define LDA_ZPAGE 0xA5
#define STA_ZPAGE 0x85
#define LDX_ZPAGE 0xA6
#define STX_ZPAGE 0x86
#define LDY_ZPAGE 0xA4
#define STY_ZPAGE 0x84
#define AND_ZPAGE 0x25
#define ORA_ZPAGE 0x05
#define EOR_ZPAGE 0x45
#define BIT_ZPAGE 0x24
#define CMP_ZPAGE 0xC5
#define CPX_ZPAGE 0xE4
#define CPY_ZPAGE 0xC4
#define ADC_ZPAGE 0x65
#define SBC_ZPAGE 0xE5
#define ASL_ZPAGE 0x06
#define LSR_ZPAGE 0x46
#define ROL_ZPAGE 0x26
#define ROR_ZPAGE 0x66
#define INC_ZPAGE 0xE6
#define DEC_ZPAGE 0xC6

#define LDA_ZPAGEX 0xB5
#define STA_ZPAGEX 0x95
#define LDY_ZPAGEX 0xB4
#define STY_ZPAGEX 0x94
#define AND_ZPAGEX 0x35
#define ORA_ZPAGEX 0x15
#define EOR_ZPAGEX 0x55
#define CMP_ZPAGEX 0xD5
#define ADC_ZPAGEX 0x75
#define SBC_ZPAGEX 0xF5
#define ASL_ZPAGEX 0x16
#define LSR_ZPAGEX 0x56
#define ROL_ZPAGEX 0x36
#define ROR_ZPAGEX 0x76
#define INC_ZPAGEX 0xF6
#define DEC_ZPAGEX 0xD6

#define LDA_ABS 0xAD
#define STA_ABS 0x8D
#define LDX_ABS 0xAE
#define STX_ABS 0x8E
#define LDY_ABS 0xAC
#define STY_ABS 0x8C
#define AND_ABS 0x2D
#define ORA_ABS 0x0D
#define EOR_ABS 0x4D
#define BIT_ABS 0x2C
#define CMP_ABS 0xCD
#define CPX_ABS 0xEC
#define CPY_ABS 0xCC
#define ADC_ABS 0x6D
#define SBC_ABS 0xED
#define ASL_ABS 0x0E
#define LSR_ABS 0x4E
#define ROL_ABS 0x2E
```

```
#define ROR_ABS 0x6E
#define INC_ABS 0xEE
#define DEC_ABS 0xCE
#define JMP_ABS 0x4C
#define JSR_ABS 0x20

#define LDA_ABSX 0xBD
#define STA_ABSX 0x9D
#define LDY_ABSX 0xBC
#define AND_ABSX 0x3D
#define ORA_ABSX 0x1D
#define EOR_ABSX 0x5D
#define CMP_ABSX 0xDD
#define ADC_ABSX 0x7D
#define SBC_ABSX 0xFD
#define ASL_ABSX 0x1E
#define LSR_ABSX 0x5E
#define ROL_ABSX 0x3E
#define ROR_ABSX 0x7E
#define INC_ABSX 0xFE
#define DEC_ABSX 0xDE

#define LDA_ABSY 0xB9
#define STA_ABSY 0x99
#define LDX_ABSY 0xBE
#define AND_ABSY 0x39
#define ORA_ABSY 0x19
#define EOR_ABSY 0x59
#define CMP_ABSY 0xD9
#define ADC_ABSY 0x79
#define SBC_ABSY 0xF9

#define LDA_INDX 0xA1
#define STA_INDX 0x81
#define AND_INDX 0x21
#define ORA_INDX 0x01
#define EOR_INDX 0x41
#define CMP_INDX 0xC1
#define ADC_INDX 0x61
#define SBC_INDX 0xE1

#define LDA_INDY 0xB1
#define STA_INDY 0x91
#define AND_INDY 0x31
#define ORA_INDY 0x11
#define EOR_INDY 0x51
#define CMP_INDY 0xD1
#define ADC_INDY 0x71
#define SBC_INDY 0xF1

#define LDX_ZPAGEY 0xB6
#define STX_ZPAGEY 0x96

#define ASL_ACCUM 0x0A
#define LSR_ACCUM 0x4A
#define ROL_ACCUM 0x2A
#define ROR_ACCUM 0x6A

#define JMP_INDIRECT_CODE1 0x6C
#define JMP_INDIRECT_CODE2 0xFF

#define BRK 0x00
#define RTS 0x60
#define RTI 0x40
#define PHP 0x08
#define PLP 0x28
#define PHA 0x48
#define PLA 0x68
#define INX 0xE8
#define DEX 0xCA
#define INY 0xC8
#define DEY 0x88
```

```
#define TAX 0xAA
#define TXA 0x8A
#define TAY 0xA8
#define TYA 0x98
#define TSX 0xBA
#define TXS 0x9A
#define SED 0xF8
#define CLD 0xD8
#define SEI 0x78
#define CLI 0x58
#define SEC 0x38
#define CLC 0x18
#define CLV 0xB8

#define BEQ 0xF0
#define BNE 0xD0
#define BMI 0x30
#define BPL 0x10
#define BCC 0x90
#define BCS 0xB0
#define BVC 0x50
#define BVS 0x70

#define NOP 0xEA
```

interrupts.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "globals.h"
#include "interrupts.h"

struct interrupts {
  Boolean IRQ;
  Boolean NMI;
  Boolean RESET;
};

Interrupts interrupts_init(void) {
  Interrupts interrupts = (Interrupts) malloc(sizeof(struct interrupts));
  assert(interrupts != NULL);

  interrupts->IRQ = FALSE;
  interrupts->NMI = FALSE;
  interrupts->RESET = FALSE;

  return interrupts;
}

void interrupts_destroy(Interrupts interrupts) {
  assert(interrupts != NULL);
  free(interrupts);
}

Boolean interrupts_getIRQ(Interrupts interrupts) {
  assert(interrupts != NULL);
  return interrupts->IRQ;
}

Boolean interrupts_getNMI(Interrupts interrupts) {
  assert(interrupts != NULL);
  return interrupts->NMI;
}

Boolean interrupts_getRESET(Interrupts interrupts) {
  assert(interrupts != NULL);
  return interrupts->RESET;
}

void interrupts_setIRQ(Interrupts interrupts, Boolean IRQ) {
  assert(interrupts != NULL);
  interrupts->IRQ = IRQ;
}

void interrupts_setNMI(Interrupts interrupts, Boolean NMI) {
  assert(interrupts != NULL);
  interrupts->NMI = NMI;
}

void interrupts_setRESET(Interrupts interrupts, Boolean RESET) {
  assert(interrupts != NULL);
  interrupts->RESET = RESET;
}
```

interrupts.h

```
#include "interrupts_type.h"

#include "globals.h"

Interrupts interrupts_init(void);

void interrupts_destroy(Interrupts interrupts);

Boolean interrupts_getIRQ(Interrupts interrupts);

Boolean interrupts_getNMI(Interrupts interrupts);

Boolean interrupts_getRESET(Interrupts interrupts);

void interrupts_setIRQ(Interrupts interrupts, Boolean IRQ);

void interrupts_setNMI(Interrupts interrupts, Boolean NMI);

void interrupts_setRESET(Interrupts interrupts, Boolean RESET);
```

interrupts_type.h

```
#ifndef INTERRUPTS_TYPE_H
  #define INTERRUPTS_TYPE_H
  typedef struct interrupts *Interrupts;
#endif
```

joypad.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "joypad.h"
#include "gui.h"
#include "nes.h"

struct joypad {
  int joypadNumber;
  Button currentButton;
};

Joypad joypad_init(int joypadNumber) {
  Joypad joypad = (Joypad) malloc(sizeof(struct joypad));
  assert(joypad != NULL);

  joypad->joypadNumber = joypadNumber;
  joypad->currentButton = 0;

  return joypad;
}

void joypad_resetCurrentButton(Joypad joypad) {
  assert(joypad != NULL);
  joypad->currentButton = 0;
}

Byte joypad_readByte(NES nes, Joypad joypad) {
  assert(nes != NULL);
  assert(joypad != NULL);

  GUI gui = nes_getGUI(nes);
  assert(gui != NULL);

  Byte data = 0;

  if (joypad->currentButton < JOYPAD_NUM_BUTTONS) {
    if (gui_isButtonPressed(gui, joypad->joypadNumber, joypad->currentButton) == TRUE) {
      data = 1;
    }
  } else if (joypad->currentButton < (JOYPAD_NUM_BUTTONS * 2)) {
    data = 0;  // player 0/2  1/3
  } else {
    data = 1;
  }

  joypad->currentButton++;

  return data;
}

void joypad_destroy(Joypad joypad) {
  assert(joypad != NULL);
  free(joypad);
}
```

joypad.h

```
#include "globals.h"

#include "nes_type.h"

#include "joypad_type.h"

#define JOYPAD_NUM_BUTTONS 8

Joypad joypad_init(int joypadNumber);

void joypad_destroy(Joypad joypad);

Byte joypad_readByte(NES nes, Joypad joypad);

void joypad_resetCurrentButton(Joypad joypad);
```

joypad_type.h

```
#ifndef JOYPAD_TYPE_H

  #define JOYPAD_TYPE_H

  typedef struct joypad *Joypad;

  typedef enum {
    JOYPAD_BUTTON_A = 0,
    JOYPAD_BUTTON_B = 1,
    JOYPAD_BUTTON_SELECT = 2,
    JOYPAD_BUTTON_START = 3,
    JOYPAD_BUTTON_UP = 4,
    JOYPAD_BUTTON_DOWN = 5,
    JOYPAD_BUTTON_LEFT = 6,
    JOYPAD_BUTTON_RIGHT = 7
  } Button;

#endif
```

memory.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "globals.h"
#include "memory.h"
#include "cpu.h"
#include "ppu.h"
#include "interrupts.h"
#include "cartridge.h"
#include "nes.h"

struct memory {
  int numAddresses;
  Byte *memory;
  NES_WriteCallback *writeCallbacks;
  NES_ReadCallback *readCallbacks;
};

Memory memory_init(int numAddresses) {
  Memory memory = malloc(sizeof(struct memory));
  assert(memory != NULL);

  memory->memory = (Byte*) malloc(sizeof(Byte) * numAddresses);
  assert(memory->memory != NULL);
  int i;
  for (i=0; i < numAddresses; i++) {
    memory->memory[i] = 0;
  }

  memory->writeCallbacks = (NES_WriteCallback*) malloc(sizeof(NES_WriteCallback) *
numAddresses);
  assert(memory->writeCallbacks != NULL);
  for (i=0; i < numAddresses; i++) {
    memory->writeCallbacks[i] = NULL;
  }

  memory->readCallbacks = (NES_ReadCallback*) malloc(sizeof(NES_ReadCallback) *
numAddresses);
  assert(memory->readCallbacks != NULL);
  for (i=0; i < numAddresses; i++) {
    memory->readCallbacks[i] = NULL;
  }

  memory->numAddresses = numAddresses;

  return memory;
}


void memory_destroy(Memory memory) {
  assert(memory != NULL);
  free(memory);
}

void memory_write_direct(Memory memory, Address address, Byte data) {
  assert(memory != NULL);
  assert(address < memory->numAddresses);

  memory->memory[address] = data;
}

Byte memory_read_direct(Memory memory, Address address) {
  assert(memory != NULL);
  assert(address < memory->numAddresses);

  return memory->memory[address];
}

void memory_print(NES nes, Memory memory) {
```

```
  assert(memory != NULL);

  debug_printf("\n");
  debug_printf("%d addresses:\n", memory->numAddresses);

  int i;
  for (i=0; i < memory->numAddresses; i++) {

    if (i % 16 == 0) {
      debug_printf("\n");
      debug_printf("%5x: ", i);
    }
    debug_printf("%02x ", memory_read_callback(nes, memory, i));
  }

  debug_printf("\n");
}

int memory_getNumAddresses(Memory memory) {
  assert(memory != NULL);
  return memory->numAddresses;
}


void memory_setWriteCallback(Memory memory, Address address, NES_WriteCallback
writeCallback) {
  assert(memory != NULL);
  memory->writeCallbacks[address] = writeCallback;
}

void memory_setReadCallback(Memory memory, Address address, NES_ReadCallback
readCallback) {
  assert(memory != NULL);
  memory->readCallbacks[address] = readCallback;
}

void memory_write_callback(NES nes, Memory memory, Address address, Byte data) {
  assert(memory != NULL);

  if (memory->writeCallbacks[address] == NULL) {
    memory_write_direct(memory, address, data);
  } else {
    memory->writeCallbacks[address](nes, address, data);
  }

}

Byte memory_read_callback(NES nes, Memory memory, Address address) {
  assert(memory != NULL);

  if (memory->readCallbacks[address] == NULL) {
    return memory_read_direct(memory, address);
  } else {
    return memory->readCallbacks[address](nes, address);
  }

}
```

memory.h

```
#include "memory_type.h"

#include "globals.h"

#include "nes_type.h"

Memory memory_init(int numAddresses);

void memory_destroy(Memory memory);

void memory_write_direct(Memory memory, Address address, Byte data);

Byte memory_read_direct(Memory memory, Address address);

void memory_print(NES nes, Memory memory);

int memory_getNumAddresses(Memory memory);

typedef Byte (*NES_ReadCallback)(NES nes, Address address);

typedef void (*NES_WriteCallback)(NES nes, Address address, Byte data);

void memory_setWriteCallback(Memory memory, Address address, NES_WriteCallback
writeCallback);

void memory_setReadCallback(Memory memory, Address address, NES_ReadCallback
readCallback);

void memory_write_callback(NES nes, Memory memory, Address address, Byte data);

Byte memory_read_callback(NES nes, Memory memory, Address address);
```

memory_type.h

```
#ifndef MEMORY_TYPE_H
  #define MEMORY_TYPE_H
  typedef struct memory *Memory;
#endif
```

mmu.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "nes.h"
#include "mmu.h"
#include "mmu0.h"
#include "cartridge.h"
#include "ppuMemory.h"
#include "memory.h"

typedef void (*MMU_CallbackCreator)(NES nes, MMU mmu);

MMU_CallbackCreator mmu_callbackCreators[] = {
  &mmu0_callbackCreator
};

int num_mmu_callbackCreators = sizeof(mmu_callbackCreators)/sizeof(MMU_CallbackCreator);

struct mmu {
  Byte activeProgramBank_lower;
  Byte activeProgramBank_upper;
  Byte activeCharacterBank;
};

MMU mmu_init(NES nes) {
  assert(nes != NULL);

  MMU mmu = (MMU) malloc(sizeof(struct mmu));
  assert(mmu != NULL);

  mmu->activeProgramBank_lower = 0;
  mmu->activeProgramBank_upper = 0;
  mmu->activeCharacterBank = 0;

  Cartridge cartridge = nes_getCartridge(nes);
  assert(cartridge != NULL);

  Byte mmuNumber = cartridge_getMMUNumber(cartridge);
  assert(mmuNumber < num_mmu_callbackCreators);

  MMU_CallbackCreator mmu_callbackCreator = mmu_callbackCreators[mmuNumber];
  assert(mmu_callbackCreator != NULL);

  mmu_callbackCreator(nes, mmu);

  return mmu;
}

Byte mmu_getActiveProgramBank_lower(MMU mmu) {
  assert(mmu != NULL);
  return mmu->activeProgramBank_lower;

}

Byte mmu_getActiveProgramBank_upper(MMU mmu) {
  assert(mmu != NULL);
  return mmu->activeProgramBank_upper;

}

Byte mmu_getActiveCharacterBank(MMU mmu) {
  assert(mmu != NULL);
  return mmu->activeCharacterBank;
}

void mmu_setActiveProgramBank_lower(MMU mmu, Byte activeProgramBank_lower) {
  assert(mmu != NULL);
  mmu->activeProgramBank_lower = activeProgramBank_lower;
}
```

```
void mmu_setActiveProgramBank_upper(MMU mmu, Byte activeProgramBank_upper) {
  assert(mmu != NULL);
  mmu->activeProgramBank_upper = activeProgramBank_upper;
}

void mmu_setActiveCharacterBank(MMU mmu, Byte activeCharacterBank) {
  assert(mmu != NULL);
  mmu->activeCharacterBank = activeCharacterBank;
}

void mmu_destroy(MMU mmu) {
  assert(mmu != NULL);
  free(mmu);
}

////////////////////////////////////////////////////////////////////////////////////////////
////////////

Byte mmu_ppuMemory_readHorizontalMirror(NES nes, Address address) {
  assert(nes != NULL);
  Memory ppuMemory = nes_getPPUMemory(nes);
  assert(ppuMemory != NULL);

  address -= PPU_NAME_TABLE_SIZE;

  return memory_read_callback(nes, ppuMemory, address);
}

void mmu_ppuMemory_writeHorizontalMirror(NES nes, Address address, Byte byte) {
  assert(nes != NULL);
  Memory ppuMemory = nes_getPPUMemory(nes);
  assert(ppuMemory != NULL);

  address -= PPU_NAME_TABLE_SIZE;

  memory_write_callback(nes, ppuMemory, address, byte);
}

////////////////////////////////////////////////////////////////////////////////////////////
////////////

Byte mmu_ppuMemory_readVerticalMirror(NES nes, Address address) {
  assert(nes != NULL);
  Memory ppuMemory = nes_getPPUMemory(nes);
  assert(ppuMemory != NULL);

  address -= PPU_NAME_TABLE_SIZE;
  address -= PPU_NAME_TABLE_SIZE;

  return memory_read_callback(nes, ppuMemory, address);
}

void mmu_ppuMemory_writeVerticalMirror(NES nes, Address address, Byte byte) {
  assert(nes != NULL);
  Memory ppuMemory = nes_getPPUMemory(nes);
  assert(ppuMemory != NULL);

  address -= PPU_NAME_TABLE_SIZE;
  address -= PPU_NAME_TABLE_SIZE;

  memory_write_callback(nes, ppuMemory, address, byte);
}

////////////////////////////////////////////////////////////////////////////////////////////
////////////
```

mmu.h

```
#include "mmu_type.h"
#include "nes_type.h"
#include "globals.h"

#define MMU_PROGRAM_BANK_LOWER_FIRST_ADDRESS 0x8000

#define MMU_PROGRAM_BANK_LOWER_LAST_ADDRESS 0xBFFF

#define MMU_PROGRAM_BANK_UPPER_FIRST_ADDRESS 0xC000

#define MMU_PROGRAM_BANK_UPPER_LAST_ADDRESS 0xFFFF

MMU mmu_init(NES nes);

Byte mmu_getActiveProgramBank_lower(MMU mmu);

Byte mmu_getActiveProgramBank_upper(MMU mmu);

Byte mmu_getActiveCharacterBank(MMU mmu);

void mmu_setActiveProgramBank_lower(MMU mmu, Byte activeProgramBank_lower);

void mmu_setActiveProgramBank_upper(MMU mmu, Byte activeProgramBank_upper);

void mmu_setActiveCharacterBank(MMU mmu, Byte activeCharacterBank);

void mmu_destroy(MMU mmu);

Byte mmu_ppuMemory_readHorizontalMirror(NES nes, Address address);

void mmu_ppuMemory_writeHorizontalMirror(NES nes, Address address, Byte byte);

Byte mmu_ppuMemory_readVerticalMirror(NES nes, Address address);

void mmu_ppuMemory_writeVerticalMirror(NES nes, Address address, Byte byte);
```

mmu_type.h

```
#ifndef MMU_TYPE_H
  #define MMU_TYPE_H
  typedef struct mmu *MMU;
#endif
```

mmu0.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "globals.h"
#include "nes.h"
#include "mmu.h"
#include "mmu0.h"
#include "memory.h"
#include "cartridge.h"
#include "ppuMemory.h"

////////////////////////////////////////////////////////////////////////////////
////////////

static Address mmu0_cpuMemory_getLocalAddress(Address address) {
  while (address >= BYTES_PER_PROGRAM_BANK) {
    address -= BYTES_PER_PROGRAM_BANK;
  }
  assert(address < BYTES_PER_PROGRAM_BANK);

  return address;
}

////////////////////////////////////////////////////////////////////////////////
////////////

static Byte mmu0_cpuMemory_readProgramBank0(NES nes, Address address) {
  assert(nes != NULL);
  //Memory memory = nes_getCPUMemory(nes);
  //assert(memory != NULL);

  address = mmu0_cpuMemory_getLocalAddress(address);

  Cartridge cartridge = nes_getCartridge(nes);
  assert(nes != NULL);

  assert(0 < cartridge_getNumProgramBanks(cartridge));

  return cartridge_readProgramBank(cartridge, 0, address);
}

static void mmu0_cpuMemory_writeProgramBank0(NES nes, Address address, Byte byte) {
  assert(nes != NULL);
  Memory memory = nes_getCPUMemory(nes);
  assert(memory != NULL);

  // not allowed to write to ROM
  assert(FALSE);
}

////////////////////////////////////////////////////////////////////////////////
////////////

static Byte mmu0_cpuMemory_readProgramBank1(NES nes, Address address) {
  assert(nes != NULL);

  address = mmu0_cpuMemory_getLocalAddress(address);

  Cartridge cartridge = nes_getCartridge(nes);
  assert(nes != NULL);

  assert(1 < cartridge_getNumProgramBanks(cartridge));

  return cartridge_readProgramBank(cartridge, 1, address);
}

static void mmu0_cpuMemory_writeProgramBank1(NES nes, Address address, Byte byte) {
  assert(nes != NULL);
  Memory memory = nes_getCPUMemory(nes);
```

```
  assert(memory != NULL);

  // not allowed to write to ROM
  assert(FALSE);
}

////////////////////////////////////////////////////////////////////////////////
////////////

static Byte mmu0_ppuMemory_readPattern(NES nes, Address address) {
  assert(nes != NULL);

  Cartridge cartridge = nes_getCartridge(nes);
  assert(cartridge != NULL);

  return cartridge_readCharacterBank(cartridge, 0, address);
}

static void mmu0_ppuMemory_writePattern(NES nes, Address address, Byte byte) {
  assert(nes != NULL);

  //assert(FALSE); // can't write to ROM
}

////////////////////////////////////////////////////////////////////////////////
////////////


void mmu0_callbackCreator(NES nes, MMU mmu) {
  assert(nes != NULL);
  assert(mmu != NULL);

  debug_printf("mmu0_callbackCreator\n");

  Memory cpuMemory = nes_getCPUMemory(nes);
  assert(cpuMemory != NULL);

  int i;
  for (i = MMU_PROGRAM_BANK_LOWER_FIRST_ADDRESS; i <=
MMU_PROGRAM_BANK_LOWER_LAST_ADDRESS; i++) {
    memory_setWriteCallback(cpuMemory, i, &mmu0_cpuMemory_writeProgramBank0);
    memory_setReadCallback(cpuMemory, i, &mmu0_cpuMemory_readProgramBank0);
  }

  Cartridge cartridge = nes_getCartridge(nes);
  assert(cartridge != NULL);

  Byte numProgramBanks = cartridge_getNumProgramBanks(cartridge);

  // mmu0 only supports 1 or 2 program banks on the cartridge
  assert(numProgramBanks >= 1);
  assert(numProgramBanks <= 2);

  if (numProgramBanks == 1) {

    // mirror the first bank if there is only one

    for (i = MMU_PROGRAM_BANK_UPPER_FIRST_ADDRESS; i <=
MMU_PROGRAM_BANK_UPPER_LAST_ADDRESS; i++) {
      memory_setWriteCallback(cpuMemory, i, &mmu0_cpuMemory_writeProgramBank0);
      memory_setReadCallback(cpuMemory, i, &mmu0_cpuMemory_readProgramBank0);
    }

  } else if (numProgramBanks == 2) {

    for (i = MMU_PROGRAM_BANK_UPPER_FIRST_ADDRESS; i <=
MMU_PROGRAM_BANK_UPPER_LAST_ADDRESS; i++) {
      memory_setWriteCallback(cpuMemory, i, &mmu0_cpuMemory_writeProgramBank1);
      memory_setReadCallback(cpuMemory, i, &mmu0_cpuMemory_readProgramBank1);
    }

  }
```

```
  Byte numCharacterBanks = cartridge_getNumCharacterBanks(cartridge);

  assert(numCharacterBanks <= 1);

  Memory ppuMemory = nes_getPPUMemory(nes);
  assert(ppuMemory != NULL);

  if (numCharacterBanks == 1) {
    // dump it in the lower space of ppu memory

    for (i = PPU_PATTERN_TABLE_0_FIRST_ADDRESS; i <= PPU_PATTERN_TABLE_1_LAST_ADDRESS;
i++) {
      memory_setWriteCallback(ppuMemory, i, &mmu0_ppuMemory_writePattern);
      memory_setReadCallback(ppuMemory, i, &mmu0_ppuMemory_readPattern);
    }

  }

  MirrorType mirrorType = cartridge_getMirrorType(cartridge);


  if (mirrorType == HORIZONTAL) {

    for (i = PPU_NAME_TABLE_1_FIRST_ADDRESS; i <= PPU_NAME_TABLE_1_LAST_ADDRESS; i++) {
      // map it back to table 0

      memory_setWriteCallback(ppuMemory, i, &mmu_ppuMemory_writeHorizontalMirror);
      memory_setReadCallback(ppuMemory, i, &mmu_ppuMemory_readHorizontalMirror);

    }

    for (i = PPU_NAME_TABLE_3_FIRST_ADDRESS; i <= PPU_NAME_TABLE_3_LAST_ADDRESS; i++) {
      // map it back to table 2

      memory_setWriteCallback(ppuMemory, i, &mmu_ppuMemory_writeHorizontalMirror);
      memory_setReadCallback(ppuMemory, i, &mmu_ppuMemory_readHorizontalMirror);

    }

  } else if (mirrorType == VERTICAL) {

    // Vertical mirroring: $2000 equals $2800 and $2400 equals $2C00 (e.g. Super Mario
Bros.)


    for (i = PPU_NAME_TABLE_2_FIRST_ADDRESS; i <= PPU_NAME_TABLE_2_LAST_ADDRESS; i++) {
      // map it back to table 0

      memory_setWriteCallback(ppuMemory, i, &mmu_ppuMemory_writeVerticalMirror);
      memory_setReadCallback(ppuMemory, i, &mmu_ppuMemory_readVerticalMirror);

    }

    for (i = PPU_NAME_TABLE_3_FIRST_ADDRESS; i <= PPU_NAME_TABLE_3_LAST_ADDRESS; i++) {
      // map it back to table 1

      memory_setWriteCallback(ppuMemory, i, &mmu_ppuMemory_writeVerticalMirror);
      memory_setReadCallback(ppuMemory, i, &mmu_ppuMemory_readVerticalMirror);

    }

  } else if (mirrorType == BOTH) {

    assert(FALSE);

  }


}
```

mmu0.h

```c
#include "nes_type.h"
#include "mmu_type.h"

void mmu0_callbackCreator(NES nes, MMU mmu);
```

nes.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "globals.h"
#include "nes.h"
#include "memory.h"
#include "cartridge.h"
#include "cpu.h"
#include "ppu.h"
#include "interrupts.h"
#include "apu.h"
#include "cpu.h"
#include "ppu.h"
#include "cpuMemory.h"
#include "ppuMemory.h"
#include "objectAttributeMemory.h"
#include "mmu.h"
#include "gui.h"
#include "joypad.h"

struct nes {
  Memory cpuMemory;
  Memory ppuMemory;
  Memory objectAttributeMemory;

  CPU cpu;
  PPU ppu;
  APU apu;

  Cartridge cartridge;

  Interrupts interrupts;

  MMU mmu;

  Boolean isRunning;

  GUI gui;

  Joypad joypads[NES_NUM_JOYPADS];
};

static void nes_init_interrupts(NES nes) {
  assert(nes != NULL);
  assert(nes->interrupts == NULL);

  nes->interrupts = interrupts_init();
  assert(nes->interrupts != NULL);

  interrupts_setRESET(nes->interrupts, TRUE);
}

static void nes_init_cpu(NES nes) {
  assert(nes != NULL);
  assert(nes->cpu == NULL);

  nes->cpu = cpu_init();
  assert(nes->cpu != NULL);
}

static void nes_init_ppu(NES nes) {
  assert(nes != NULL);
  assert(nes->ppu == NULL);

  nes->ppu = ppu_init();
  assert(nes->ppu != NULL);
}

static void nes_init_apu(NES nes) {
```

```
  assert(nes != NULL);
  assert(nes->apu == NULL);

  nes->apu = apu_init();
  assert(nes->apu != NULL);
}

static void nes_init_cartridge(NES nes, char *filename) {
  assert(nes != NULL);
  assert(filename != NULL);
  assert(nes->cartridge == NULL);

  nes->cartridge = cartridge_init(filename);
  assert(nes->cartridge != NULL);
}

static void nes_init_ppuMemory(NES nes) {
  assert(nes != NULL);
  assert(nes->ppuMemory == NULL);

  nes->ppuMemory = ppuMemory_init();
  assert(nes->ppuMemory != NULL);
}

static void nes_init_cpuMemory(NES nes) {
  assert(nes != NULL);
  assert(nes->cpuMemory == NULL);

  nes->cpuMemory = cpuMemory_init();
  assert(nes->cpuMemory != NULL);
}

static void nes_init_mmu(NES nes) {
  assert(nes != NULL);
  assert(nes->mmu == NULL);

  nes->mmu = mmu_init(nes);
  assert(nes->mmu != NULL);
}

static void nes_init_objectAttributeMemory(NES nes) {
  assert(nes != NULL);
  assert(nes->objectAttributeMemory == NULL);

  nes->objectAttributeMemory = objectAttributeMemory_init();
  assert(nes->objectAttributeMemory != NULL);
}

Memory nes_getObjectAttributeMemory(NES nes) {
  assert(nes != NULL);
  assert(nes->objectAttributeMemory != NULL);
  return nes->objectAttributeMemory;
}

void nes_generateNMI(NES nes) {
  assert(nes != NULL);
  debug_printf("generateNMI\n");
  interrupts_setNMI(nes->interrupts, TRUE);
}

static void nes_init_gui(NES nes, int width, int height) {
  assert(nes != NULL);
  assert(nes->gui == NULL);
  nes->gui = gui_init(width, height);
  assert(nes->gui != NULL);
}

static void nes_init_joypads(NES nes) {
  assert(nes != NULL);

  int i;
  for (i=0; i < NES_NUM_JOYPADS; i++) {
```

```
    nes->joypads[i] = joypad_init(i);
    assert(nes->joypads[i] != NULL);
  }
}

NES nes_init(char *filename, int width, int height) {
  assert(filename != NULL);

  NES nes = (NES) malloc(sizeof(struct nes));
  assert(nes != NULL);

  nes->cartridge = NULL;
  nes->ppuMemory = NULL;
  nes->cpuMemory = NULL;
  nes->ppu = NULL;
  nes->cpu = NULL;
  nes->apu = NULL;
  nes->interrupts = NULL;
  nes->mmu = NULL;
  nes->objectAttributeMemory = NULL;
  nes->gui = NULL;

  nes_init_cpuMemory(nes);
  nes_init_ppuMemory(nes);
  nes_init_cartridge(nes, filename);
  nes_init_apu(nes);
  nes_init_cpu(nes);
  nes_init_ppu(nes);
  nes_init_interrupts(nes);
  nes_init_objectAttributeMemory(nes);
  nes_init_gui(nes, width, height);
  nes_init_joypads(nes);

  // MMU should be the last thing init, since it can create callbacks all over the place
  nes_init_mmu(nes);

  debug_printf("\n");
  debug_printf("Initial cpuMemory\n");

  memory_print(nes, nes->cpuMemory);

  debug_printf("\n");
  debug_printf("Initial ppuMemory\n");

  memory_print(nes, nes->ppuMemory);

  debug_printf("\n");
  debug_printf("Initial objectAttributeMemory\n");

  memory_print(nes, nes->objectAttributeMemory);

  nes->isRunning = TRUE;

  return nes;
}

Byte nes_readJoypad(NES nes, int joypadNumber) {
  assert(nes != NULL);
  assert(joypadNumber < NES_NUM_JOYPADS);

  return joypad_readByte(nes, nes->joypads[joypadNumber]);
}

void nes_writeJoypad(NES nes, int joypadNumber, Byte data) {
  assert(nes != NULL);
  assert(joypadNumber < NES_NUM_JOYPADS);

  if (joypadNumber == 0) {

    if ((data & MASK_BIT0) == MASK_BIT0) {
      gui_queryInput(nes->gui);
    } else {
```

```
      int i;
      for (i=0; i < NES_NUM_JOYPADS; i++) {
        joypad_resetCurrentButton(nes->joypads[i]);
      }
    }

  }
}

void nes_destroy(NES nes) {
  assert(nes != NULL);

  memory_destroy(nes->cpuMemory);
  memory_destroy(nes->ppuMemory);
  memory_destroy(nes->objectAttributeMemory);

  cpu_destroy(nes->cpu);
  ppu_destroy(nes->ppu);
  apu_destroy(nes->apu);

  cartridge_destroy(nes->cartridge);

  interrupts_destroy(nes->interrupts);

  mmu_destroy(nes->mmu);

  gui_destroy(nes->gui);

  int i;
  for (i=0; i < NES_NUM_JOYPADS; i++) {
    joypad_destroy(nes->joypads[i]);
  }

  free(nes);
}

static void nes_checkKeyboard(NES nes) {
  assert(nes != NULL);

  nes->isRunning = !gui_receivedTerminationRequest(nes->gui);
}

void nes_run(NES nes) {
  assert(nes != NULL);

  while(nes->isRunning == TRUE) {

    debug_printf("nes_run\n");

    if (interrupts_getRESET(nes->interrupts) == TRUE) {

      debug_printf("RESET\n");
      interrupts_setRESET(nes->interrupts, FALSE);

      cpu_handleInterrupt(nes, CPU_RESET_VECTOR_LOWER_ADDRESS, FALSE);

    } else if (interrupts_getNMI(nes->interrupts) == TRUE) {

      debug_printf("NMI\n");
      interrupts_setNMI(nes->interrupts, FALSE);

      cpu_handleInterrupt(nes, CPU_NMI_VECTOR_LOWER_ADDRESS, FALSE);

    } else if ((cpu_getInterruptDisable(nes->cpu) == FALSE) && (interrupts_getIRQ(nes-
>interrupts) == TRUE)) {

      debug_printf("IRQ\n");
      interrupts_setIRQ(nes->interrupts, FALSE);

      cpu_handleInterrupt(nes, CPU_IRQ_VECTOR_LOWER_ADDRESS, FALSE);
```

```
    }

    cpu_step(nes);

    nes_checkKeyboard(nes);
  }
}

GUI nes_getGUI(NES nes) {
  assert(nes != NULL);
  assert(nes->gui != NULL);
  return nes->gui;
}

Memory nes_getCPUMemory(NES nes) {
  assert(nes != NULL);
  return nes->cpuMemory;
}

Memory nes_getPPUMemory(NES nes) {
  assert(nes != NULL);
  return nes->ppuMemory;
}

CPU nes_getCPU(NES nes) {
  assert(nes != NULL);
  return nes->cpu;
}

PPU nes_getPPU(NES nes) {
  assert(nes != NULL);
  return nes->ppu;
}

Cartridge nes_getCartridge(NES nes) {
  assert(nes != NULL);
  return nes->cartridge;
}

Interrupts nes_getInterrupts(NES nes) {
  assert(nes != NULL);
  return nes->interrupts;
}

Byte nes_readCPUMemory(NES nes, Address address) {
  assert(nes != NULL);
  assert(nes->cpuMemory != NULL);

  Byte data = memory_read_callback(nes, nes->cpuMemory, address);

  nes_cpuCycled(nes);

  return data;
}

void nes_writeCPUMemory(NES nes, Address address, Byte data) {
  assert(nes != NULL);
  assert(nes->cpuMemory != NULL);

  memory_write_callback(nes, nes->cpuMemory, address, data);

  nes_cpuCycled(nes);
}


Byte nes_readObjectAttributeMemory(NES nes, Address address) {
  assert(nes != NULL);
  assert(nes->objectAttributeMemory != NULL);

  return memory_read_callback(nes, nes->objectAttributeMemory, address);
}
```

```
void nes_writeObjectAttributeMemory(NES nes, Address address, Byte data) {
  assert(nes != NULL);
  assert(nes->objectAttributeMemory != NULL);

  memory_write_callback(nes, nes->objectAttributeMemory, address, data);
}

// the PPU has 64kb of address space but can only access 16kb of it
// we handle the "big picture" ppu mirrors here (the four 16kb mirrors)
// and we handle the smaller mirrors (mirrors within mirrors) with callbacks
static Address nes_getLowPPUAddress(Address address) {

  while(address > PPU_LAST_REAL_ADDRESS) {
    address -= PPU_NUM_REAL_ADDRESSES;
  }

  assert(address <= PPU_LAST_REAL_ADDRESS);

  return address;
}

Byte nes_readPPUMemory(NES nes, Address address) {
  assert(nes != NULL);

  address = nes_getLowPPUAddress(address);

  return memory_read_callback(nes, nes_getPPUMemory(nes), address);
}

void nes_writePPUMemory(NES nes, Address address, Byte data) {
  assert(nes != NULL);

  address = nes_getLowPPUAddress(address);

  memory_write_callback(nes, nes_getPPUMemory(nes), address, data);
}

APU getAPU(NES nes) {
  assert(nes != NULL);
  return nes->apu;
}

void nes_cpuCycled(NES nes) {
  assert(nes != NULL);

  ppu_step(nes);
  apu_step(nes);

  ppu_step(nes);
  apu_step(nes);

  ppu_step(nes);
  apu_step(nes);

  // adjust for pal frequency here
  // if (PAL) {
  // }
}
```

nes.h

```
#include "nes_type.h"
#include "globals.h"
#include "memory_type.h"
#include "cpu_type.h"
#include "ppu_type.h"
#include "cartridge_type.h"
#include "interrupts_type.h"
#include "apu_type.h"
#include "gui.h"

#define NES_NUM_JOYPADS 2

NES nes_init(char *filename, int width, int height);

void nes_destroy(NES nes);

Memory nes_getCPUMemory(NES nes);
Memory nes_getPPUMemory(NES nes);
Memory nes_getObjectAttributeMemory(NES nes);

CPU nes_getCPU(NES nes);
PPU nes_getPPU(NES nes);
APU nes_getAPU(NES nes);
Cartridge nes_getCartridge(NES nes);
Interrupts nes_getInterrupts(NES nes);

void nes_run(NES nes);

Byte nes_readCPUMemory(NES nes, Address address);
void nes_writeCPUMemory(NES nes, Address address, Byte data);

Byte nes_readPPUMemory(NES nes, Address address);
void nes_writePPUMemory(NES nes, Address address, Byte data);

Byte nes_readObjectAttributeMemory(NES nes, Address address);
void nes_writeObjectAttributeMemory(NES nes, Address address, Byte data);

void nes_cpuCycled(NES nes);

void nes_generateNMI(NES nes);

GUI nes_getGUI(NES nes);

Byte nes_readJoypad(NES nes, int joypadNumber);

void nes_writeJoypad(NES nes, int joypadNumber, Byte data);
```

nes_type.h

```
#ifndef NES_TYPE_H
  #define NES_TYPE_H
  typedef struct nes *NES;
#endif
```

objectAttributeMemory.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "memory.h"
#include "objectAttributeMemory.h"
#include "globals.h"

/*
       There is an independent area of memory known as sprite ram, which is
256 bytes in size.  This memory stores 4 bytes of information for 64 sprites.
These 4 bytes contain the x and y location of the sprite on the screen, the
upper two color bits, the tile index number (pattern table tile of the sprite),
and information about flipping (horizontal and vertical), and priority
(behind/on top of background).  Sprite ram can be accessed byte-by-byte through
the NES registers, or also can be loaded via DMA transfer through another
register.
*/

// assumption: not using callbacks within sprite memory?

Memory objectAttributeMemory_init(void) {
  Memory memory = memory_init(OAM_NUM_ADDRESSES);
  assert(memory != NULL);
  return memory;
}

// byte 0
// 0    -       scanline coordinate minus one of object's top pixel row.
Byte objectAttributeMemory_getY(Memory memory, int spriteIndex) {
  assert(memory != NULL);

  Byte address = (OAM_BYTES_PER_SPRITE * spriteIndex) + OAM_Y_BYTE_OFFSET;

  return memory_read_direct(memory, address);
}

// byte 1

// only valid for 8x16 sprites
Byte objectAttributeMemory_getBankNumber(Memory memory, int spriteIndex) {
  assert(memory != NULL);

  Byte address = (OAM_BYTES_PER_SPRITE * spriteIndex) + OAM_TILE_BYTE_OFFSET;

  Byte data = memory_read_direct(memory, address);

  if ( (data & MASK_OAM_BANK_NUMBER_ON) == MASK_OAM_BANK_NUMBER_ON) {
    return 1;
  } else {
    return 0;
  }
}

Byte objectAttributeMemory_getTileNumber(Memory memory, int spriteIndex, Boolean
using8x16) {
  assert(memory != NULL);

  Byte address = (OAM_BYTES_PER_SPRITE * spriteIndex) + OAM_TILE_BYTE_OFFSET;
  Byte data = memory_read_direct(memory, address);

  if (using8x16 == FALSE) {
    return data;
  } else {
    // lose the 0th bit
    return (data >> 1) << 1;
  }
}

// byte 2
```

```
Byte objectAttributeMemory_getAttributes(Memory memory, int spriteIndex) {
  assert(memory != NULL);

  Byte address = (OAM_BYTES_PER_SPRITE * spriteIndex) + OAM_ATTRIBUTES_BYTE_OFFSET;

  return memory_read_direct(memory, address);
}

// Palette (4 to 7) of sprite
Byte objectAttributeMemory_getPalette(Memory memory, int spriteIndex) {
  assert(memory != NULL);

  Byte attributes = objectAttributeMemory_getAttributes(memory, spriteIndex);

  // lose the 765432 bits
  attributes = (attributes << 6) >> 4;

  return attributes; // + 4;
}

Boolean objectAttributeMemory_isBehindBackground(Memory memory, int spriteIndex) {
  assert(memory != NULL);

  Byte attributes = objectAttributeMemory_getAttributes(memory, spriteIndex);

  if ( (attributes & MASK_OAM_BEHIND_BACKGROUND_ON) == MASK_OAM_BEHIND_BACKGROUND_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

Boolean objectAttributeMemory_isFlippedHorizontal(Memory memory, int spriteIndex) {
  assert(memory != NULL);

  Byte attributes = objectAttributeMemory_getAttributes(memory, spriteIndex);

  if ( (attributes & MASK_OAM_FLIP_HORIZONTAL_ON) == MASK_OAM_FLIP_HORIZONTAL_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

Boolean objectAttributeMemory_isFlippedVertical(Memory memory, int spriteIndex) {
  assert(memory != NULL);

  Byte attributes = objectAttributeMemory_getAttributes(memory, spriteIndex);

  if ( (attributes & MASK_OAM_FLIP_VERTICAL_ON) == MASK_OAM_FLIP_VERTICAL_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

// byte 3
// 3    -       scanline pixel coordite of most left-hand side of object.
Byte objectAttributeMemory_getX(Memory memory, int spriteIndex) {
  assert(memory != NULL);

  Byte address = (OAM_BYTES_PER_SPRITE * spriteIndex) + OAM_X_BYTE_OFFSET;

  return memory_read_direct(memory, address);
}
```

objectAttributeMemory.h

```
#include "globals.h"

#include "memory_type.h"

#define OAM_FIRST_ADDRESS 0
#define OAM_LAST_ADDRESS 255

#define OAM_NUM_ADDRESSES (OAM_LAST_ADDRESS + 1)

#define OAM_BYTES_PER_SPRITE 4

#define OAM_NUMBER_OF_SPRITES (OAM_NUM_ADDRESSES / OAM_BYTES_PER_SPRITE)

#define OAM_BYTES_PER_SPRITE 4

#define OAM_Y_BYTE_OFFSET 0
#define OAM_TILE_BYTE_OFFSET 1
#define OAM_ATTRIBUTES_BYTE_OFFSET 2
#define OAM_X_BYTE_OFFSET 3

#define MASK_OAM_FLIP_HORIZONTAL_ON MASK_BIT6
#define MASK_OAM_FLIP_VERTICAL_ON MASK_BIT7
#define MASK_OAM_BEHIND_BACKGROUND_ON MASK_BIT5
#define MASK_OAM_BANK_NUMBER_ON MASK_BIT0

Memory objectAttributeMemory_init(void);


// byte 0
Byte objectAttributeMemory_getY(Memory memory, int spriteIndex);

// byte 1

Byte objectAttributeMemory_getTileNumber(Memory memory, int spriteIndex, Boolean
using8x16);

// only valid for 8x16 sprites
Byte objectAttributeMemory_getBankNumber(Memory memory, int spriteIndex);

// byte 2

Byte objectAttributeMemory_getAttributes(Memory memory, int spriteIndex);

Byte objectAttributeMemory_getPalette(Memory memory, int spriteIndex);

Boolean objectAttributeMemory_isBehindBackground(Memory memory, int spriteIndex);

Boolean objectAttributeMemory_isFlippedHorizontal(Memory memory, int spriteIndex);

Boolean objectAttributeMemory_isFlippedVertical(Memory memory, int spriteIndex);

// byte 3

Byte objectAttributeMemory_getX(Memory memory, int spriteIndex);
```

ppu.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "globals.h"
#include "ppu.h"
#include "nes.h"
#include "ppuMemory.h"
#include "colour.h"
#include "objectAttributeMemory.h"

// palette adapted from http://nesdev.parodius.com/NESTechFAQ.htm

struct colour systemPalette[PPU_NUM_SYSTEM_COLOURS] = {
  {0x75, 0x75, 0x75},
  {0x27, 0x1B, 0x8F},
  {0x00, 0x00, 0xAB},
  {0x47, 0x00, 0x9F},
  {0x8F, 0x00, 0x77},
  {0xAB, 0x00, 0x13},
  {0xA7, 0x00, 0x00},
  {0x7F, 0x0B, 0x00},
  {0x43, 0x2F, 0x00},
  {0x00, 0x47, 0x00},
  {0x00, 0x51, 0x00},
  {0x00, 0x3F, 0x17},
  {0x1B, 0x3F, 0x5F},
  {0x00, 0x00, 0x00},
  {0x00, 0x00, 0x00},
  {0x00, 0x00, 0x00},
  {0xBC, 0xBC, 0xBC},
  {0x00, 0x73, 0xEF},
  {0x23, 0x3B, 0xEF},
  {0x83, 0x00, 0xF3},
  {0xBF, 0x00, 0xBF},
  {0xE7, 0x00, 0x5B},
  {0xDB, 0x2B, 0x00},
  {0xCB, 0x4F, 0x0F},
  {0x8B, 0x73, 0x00},
  {0x00, 0x97, 0x00},
  {0x00, 0xAB, 0x00},
  {0x00, 0x93, 0x3B},
  {0x00, 0x83, 0x8B},
  {0x00, 0x00, 0x00},
  {0x00, 0x00, 0x00},
  {0x00, 0x00, 0x00},
  {0xFF, 0xFF, 0xFF},
  {0x3F, 0xBF, 0xFF},
  {0x5F, 0x97, 0xFF},
  {0xA7, 0x8B, 0xFD},
  {0xF7, 0x7B, 0xFF},
  {0xFF, 0x77, 0xB7},
  {0xFF, 0x77, 0x63},
  {0xFF, 0x9B, 0x3B},
  {0xF3, 0xBF, 0x3F},
  {0x83, 0xD3, 0x13},
  {0x4F, 0xDF, 0x4B},
  {0x58, 0xF8, 0x98},
  {0x00, 0xEB, 0xDB},
  {0x00, 0x00, 0x00},
  {0x00, 0x00, 0x00},
  {0x00, 0x00, 0x00},
  {0xFF, 0xFF, 0xFF},
  {0xAB, 0xE7, 0xFF},
  {0xC7, 0xD7, 0xFF},
  {0xD7, 0xCB, 0xFF},
  {0xFF, 0xC7, 0xFF},
  {0xFF, 0xC7, 0xDB},
  {0xFF, 0xBF, 0xB3},
  {0xFF, 0xDB, 0xAB},
```

```
  {0xFF, 0xE7, 0xA3},
  {0xE3, 0xFF, 0xA3},
  {0xAB, 0xF3, 0xBF},
  {0xB3, 0xFF, 0xCF},
  {0x9F, 0xFF, 0xF3},
  {0x00, 0x00, 0x00},
  {0x00, 0x00, 0x00},
  {0x00, 0x00, 0x00}
};

Byte
attributeTableLookup[PPU_VERTICAL_TILES_PER_ATTRIBUTE_BYTE][PPU_HORIZONTAL_TILES_PER_ATT
RIBUTE_BYTE] = {
  {0x0, 0x1, 0x4, 0x5},
  {0x2, 0x3, 0x6, 0x7},
  {0x8, 0x9, 0xC, 0xD},
  {0xA, 0xB, 0xE, 0xF}
};

struct ppu {

  // the 8 exposed registers
  Byte controlRegister;
  Byte maskRegister;
  Byte statusRegister;
  Byte spriteAddressRegister;
  Byte spriteDataRegister;   // unused?
  Byte scrollRegister;
  Byte ppuMemoryAddressRegister;
  Byte ppuMemoryDataRegister;

  // book-keeping

  Byte ppuMemoryReadBuffer;

  Byte verticalScroll;      // Vertical offsets range from -16 to 239
                            // (values greater than 240 are treated as negative, though
they actually render their tile data from the attribute table).
  Byte horizontalScroll;    // Horizontal offsets range from 0 to 255;

  Boolean hasPartial;
  Byte ppuAddressLow;
  Byte ppuAddressHigh;

  int currentCycle;
  int currentScanline;
  int currentScanlineCycle;
  int currentFrame;

  Byte spriteColoursForScanline[PPU_SCREEN_WIDTH_IN_PIXELS];
  Boolean spriteColoursForScanlineSet[PPU_SCREEN_WIDTH_IN_PIXELS];
  Boolean spriteColoursForScanlineIsBehindBackground[PPU_SCREEN_WIDTH_IN_PIXELS];

};

static void ppu_resetSpriteColoursForScanline(PPU ppu) {
  assert(ppu != NULL);

  int i;
  for (i=0; i < PPU_SCREEN_WIDTH_IN_PIXELS; i++) {
    ppu->spriteColoursForScanline[i] = 0;
    ppu->spriteColoursForScanlineSet[i] = FALSE;
    ppu->spriteColoursForScanlineIsBehindBackground[i] = FALSE;
  }
}

PPU ppu_init(void) {
  PPU ppu = (PPU) malloc(sizeof(struct ppu));
  assert(ppu != NULL);

  ppu->controlRegister = 0;
  ppu->maskRegister = 0;
```

```
  ppu->statusRegister = 0;
  ppu->spriteAddressRegister = 0;
  ppu->spriteDataRegister = 0;    // unused?

  ppu->ppuAddressLow = 0;
  ppu->ppuAddressHigh = 0;
  ppu->ppuMemoryAddressRegister = 0;
  ppu->ppuMemoryDataRegister = 0;
  ppu->hasPartial = FALSE;
  ppu->verticalScroll = 0;
  ppu->horizontalScroll = 0;
  ppu->scrollRegister = 0;

  ppu->currentCycle = 0;

  ppu->currentScanline = 0;

  ppu->currentScanlineCycle = 0;

  ppu_resetSpriteColoursForScanline(ppu);

  ppu->ppuMemoryReadBuffer = 0;

  ppu->currentFrame = 0;

  return ppu;
}

void ppu_destroy(PPU ppu) {
  assert(ppu != NULL);
  free(ppu);
}

static Boolean ppu_getControlHorizontalScrollNametable(PPU ppu) {
  assert(ppu != NULL);

  if ( (ppu->controlRegister & MASK_CONTROL_HORIZONTAL_SCROLL_NAME_TABLE_ON) ==
MASK_CONTROL_HORIZONTAL_SCROLL_NAME_TABLE_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void ppu_setControlHorizontalScrollNametable(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->controlRegister |= MASK_CONTROL_HORIZONTAL_SCROLL_NAME_TABLE_ON;
  } else if (state == FALSE) {
    ppu->controlRegister &= MASK_CONTROL_HORIZONTAL_SCROLL_NAME_TABLE_OFF;
  }
}

static Boolean ppu_getControlVerticalScrollNametable(PPU ppu) {
  assert(ppu != NULL);

  if ( (ppu->controlRegister & MASK_CONTROL_VERTICAL_SCROLL_NAME_TABLE_ON) ==
MASK_CONTROL_VERTICAL_SCROLL_NAME_TABLE_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void ppu_setControlVerticalScrollNametable(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->controlRegister |= MASK_CONTROL_VERTICAL_SCROLL_NAME_TABLE_ON;
  } else if (state == FALSE) {
    ppu->controlRegister &= MASK_CONTROL_VERTICAL_SCROLL_NAME_TABLE_OFF;
```

```
  }
}

static Address ppu_getCurrentBaseNametableAddress(PPU ppu) {
  assert(ppu != NULL);

  Address address = PPU_NAME_TABLE_0_FIRST_ADDRESS;

  if (ppu_getControlHorizontalScrollNametable(ppu) == TRUE) {
    address += PPU_NAME_TABLE_SIZE;
  }


  if (ppu_getControlVerticalScrollNametable(ppu) == TRUE) {
    address += PPU_NAME_TABLE_SIZE;
    address += PPU_NAME_TABLE_SIZE;
  }


  return address;
}

static Boolean cpu_getMaskIntensifyRed(PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->maskRegister & MASK_MASK_INTENSIFY_RED_ON) == MASK_MASK_INTENSIFY_RED_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void cpu_setMarkIntensifyRed(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->maskRegister |= MASK_MASK_INTENSIFY_RED_ON;
  } else if (state == FALSE) {
    ppu->maskRegister &= MASK_MASK_INTENSIFY_RED_OFF;
  }
}

static Boolean cpu_getMaskIntensifyGren(PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->maskRegister & MASK_MASK_INTENSIFY_GREEN_ON) ==
MASK_MASK_INTENSIFY_GREEN_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void cpu_setMarkIntensifyGreen(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->maskRegister |= MASK_MASK_INTENSIFY_GREEN_ON;
  } else if (state == FALSE) {
    ppu->maskRegister &= MASK_MASK_INTENSIFY_GREEN_OFF;
  }
}

static Boolean cpu_getMaskIntensifyBlue(PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->maskRegister & MASK_MASK_INTENSIFY_BLUE_ON) == MASK_MASK_INTENSIFY_BLUE_ON)
{
    return TRUE;
  } else {
    return FALSE;
  }
```

```
}

static void cpu_setMarkIntensifyBlue(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->maskRegister |= MASK_MASK_INTENSIFY_BLUE_ON;
  } else if (state == FALSE) {
    ppu->maskRegister &= MASK_MASK_INTENSIFY_BLUE_OFF;
  }
}

static void ppu_setControlPPUAddressIncrement(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->controlRegister |= MASK_CONTROL_PPU_ADDRESS_INCREMENT_ON;
  } else if (state == FALSE) {
    ppu->controlRegister &= MASK_CONTROL_PPU_ADDRESS_INCREMENT_OFF;
  }
}

static Boolean ppu_getControlPPUAddressIncrement(PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->controlRegister & MASK_CONTROL_PPU_ADDRESS_INCREMENT_ON) ==
MASK_CONTROL_PPU_ADDRESS_INCREMENT_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void ppu_setControlSpriteTileTable(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->controlRegister |= MASK_CONTROL_SPRITE_TILE_TABLE_ON;
  } else if (state == FALSE) {
    ppu->controlRegister &= MASK_CONTROL_SPRITE_TILE_TABLE_OFF;
  }
}

static Boolean ppu_getControlSpriteTileTable(PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->controlRegister & MASK_CONTROL_SPRITE_TILE_TABLE_ON) ==
MASK_CONTROL_SPRITE_TILE_TABLE_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void ppu_setControlBackgroundTileTable(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->controlRegister |= MASK_CONTROL_BACKGROUND_TILE_TABLE_ON;
  } else if (state == FALSE) {
    ppu->controlRegister &= MASK_CONTROL_BACKGROUND_TILE_TABLE_OFF;
  }
}

static Boolean ppu_getControlBackgroundTileTable(PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->controlRegister & MASK_CONTROL_BACKGROUND_TILE_TABLE_ON) ==
MASK_CONTROL_BACKGROUND_TILE_TABLE_ON) {
    return TRUE;
  } else {
    return FALSE;
```

```
  }
}

static void ppu_setControlSpriteSize(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->controlRegister |= MASK_CONTROL_SPRITE_SIZE_ON;
  } else if (state == FALSE) {
    ppu->controlRegister &= MASK_CONTROL_SPRITE_SIZE_OFF;
  }
}

static Boolean ppu_getControlSpriteSize(PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->controlRegister & MASK_CONTROL_SPRITE_SIZE_ON) ==
MASK_CONTROL_SPRITE_SIZE_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void ppu_setControlNMI(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->controlRegister |= MASK_CONTROL_NMI_ON;
  } else if (state == FALSE) {
    ppu->controlRegister &= MASK_CONTROL_NMI_OFF;
  }
}

static Boolean ppu_getControlNMI(PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->controlRegister & MASK_CONTROL_NMI_ON) == MASK_CONTROL_NMI_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void ppu_setMaskDisplayType(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->maskRegister |= MASK_MASK_DISPLAY_TYPE_ON;
  } else if (state == FALSE) {
    ppu->maskRegister &= MASK_MASK_DISPLAY_TYPE_OFF;
  }
}

static Boolean ppu_getMaskDisplayType(PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->maskRegister & MASK_MASK_DISPLAY_TYPE_ON) == MASK_MASK_DISPLAY_TYPE_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void ppu_setMaskBackgroundClipping(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->maskRegister |= MASK_MASK_BACKGROUND_CLIPPING_ON;
  } else if (state == FALSE) {
    ppu->maskRegister &= MASK_MASK_BACKGROUND_CLIPPING_OFF;
  }
```

```
}

static Boolean ppu_getMaskBackgroundClipping(PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->maskRegister & MASK_MASK_BACKGROUND_CLIPPING_ON) ==
MASK_MASK_BACKGROUND_CLIPPING_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void ppu_setMaskSpriteClipping(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->maskRegister |= MASK_MASK_SPRITE_CLIPPING_ON;
  } else if (state == FALSE) {
    ppu->maskRegister &= MASK_MASK_SPRITE_CLIPPING_OFF;
  }
}

static Boolean ppu_getMaskSpriteClipping(PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->maskRegister & MASK_MASK_SPRITE_CLIPPING_ON) ==
MASK_MASK_SPRITE_CLIPPING_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void ppu_setMaskBackgroundVisiblity(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->maskRegister |= MASK_MASK_BACKGROUND_VISIBILITY_ON;
  } else if (state == FALSE) {
    ppu->maskRegister &= MASK_MASK_BACKGROUND_VISIBILITY_OFF;
  }
}

static Boolean ppu_getMaskBackgroundVisibility (PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->maskRegister & MASK_MASK_BACKGROUND_VISIBILITY_ON) ==
MASK_MASK_BACKGROUND_VISIBILITY_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void ppu_setMaskSpriteVisibility(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->maskRegister |= MASK_MASK_SPRITE_VISIBILITY_ON;
  } else if (state == FALSE) {
    ppu->maskRegister &= MASK_MASK_SPRITE_VISIBILITY_OFF;
  }
}

static Boolean ppu_getMaskSpriteVisibility(PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->maskRegister & MASK_MASK_SPRITE_VISIBILITY_ON) ==
MASK_MASK_SPRITE_VISIBILITY_ON) {
    return TRUE;
  } else {
```

```
    return FALSE;
  }
}


static void ppu_setStatusSpriteOverflow(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->statusRegister |= MASK_STATUS_SPRITE_OVERFLOW_ON;
  } else if (state == FALSE) {
    ppu->statusRegister &= MASK_STATUS_SPRITE_OVERFLOW_OFF;
  }
}

static Boolean ppu_getStatusSpriteOverflow(PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->statusRegister & MASK_STATUS_SPRITE_OVERFLOW_ON) ==
MASK_STATUS_SPRITE_OVERFLOW_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void ppu_setStatusSpriteCollisionHit(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->statusRegister |= MASK_STATUS_SPRITE_COLLISION_HIT_ON;
  } else if (state == FALSE) {
    ppu->statusRegister &= MASK_STATUS_SPRITE_COLLISION_HIT_OFF;
  }
}

static Boolean ppu_getStatusSpriteCollisionHit(PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->statusRegister & MASK_STATUS_SPRITE_COLLISION_HIT_ON) ==
MASK_STATUS_SPRITE_COLLISION_HIT_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}

static void ppu_setStatusVerticalBlank(PPU ppu, Boolean state) {
  assert(ppu != NULL);

  if (state == TRUE) {
    ppu->statusRegister |= MASK_STATUS_VBLANK_ON;
  } else if (state == FALSE) {
    ppu->statusRegister &= MASK_STATUS_VBLANK_OFF;
  }
}

static Boolean ppu_getStatusVerticalBlank(PPU ppu) {
  assert(ppu != NULL);

  if ((ppu->statusRegister & MASK_STATUS_VBLANK_ON) == MASK_STATUS_VBLANK_ON) {
    return TRUE;
  } else {
    return FALSE;
  }
}


static Byte ppu_getCurrentX(PPU ppu) {
  assert(ppu != NULL);
  return ppu->currentScanlineCycle;
}
```

```
static Byte ppu_getCurrentY(PPU ppu) {
  assert(ppu != NULL);
  return ppu->currentScanline - PPU_WASTED_VBLANK_SCANLINES -
PPU_WASTED_PREFETCH_SCANLINES;
}

static Word ppu_getBackgroundTileNumber(Word x, Word y) {
  Word horizontalOffset = (x / PPU_HORIZONTAL_PIXELS_PER_TILE);
  Word verticalOffset = (y / PPU_VERTICAL_PIXELS_PER_TILE) *
PPU_BACKGROUND_TILES_PER_ROW;

  Word tileNumber = horizontalOffset + verticalOffset;

  return tileNumber;
}

static Byte ppu_getBackgroundNametableByte(NES nes) {
  assert(nes != NULL);
  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  Address nametableStart = ppu_getCurrentBaseNametableAddress(ppu);
  assert(nametableStart == PPU_NAME_TABLE_0_FIRST_ADDRESS ||
         nametableStart == PPU_NAME_TABLE_1_FIRST_ADDRESS ||
         nametableStart == PPU_NAME_TABLE_2_FIRST_ADDRESS ||
         nametableStart == PPU_NAME_TABLE_3_FIRST_ADDRESS);

  Word x = ppu_getCurrentX(ppu);
  Word y = ppu_getCurrentY(ppu);

  Word tileNumber = ppu_getBackgroundTileNumber(x, y);

  Address nametableByteAddress = nametableStart + tileNumber;
  VALIDATE_NAMETABLE_ADDRESS(nametableByteAddress);

  return nes_readPPUMemory(nes,nametableByteAddress);
}

static Byte ppu_combinePatternBytes(PPU ppu, Byte pattern1, Byte pattern2, Byte x) {
  assert(ppu != NULL);

  x %= PPU_HORIZONTAL_PIXELS_PER_TILE;

  pattern1 = pattern1 << x;
  pattern1 = pattern1 >> (PPU_HORIZONTAL_PIXELS_PER_TILE - 1);

  pattern2 = pattern2 << x;
  pattern2 = pattern2 >> (PPU_HORIZONTAL_PIXELS_PER_TILE - 1);
  pattern2 = pattern2 << 1;

  Byte patternIndex = pattern1 + pattern2;
  assert(patternIndex <= 3);

  return patternIndex;
}


static Byte ppu_getPatternColourIndex(NES nes, Address basePatternAddress, Byte
patternTileNumber, Byte x, Byte y) {
  assert(nes != NULL);
  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  Word horizontalOffset = patternTileNumber * PPU_PATTERN_BYTES_PER_TILE;

  Word verticalOffset = (y % PPU_VERTICAL_PIXELS_PER_TILE);

  Address pattern1Address = basePatternAddress + horizontalOffset + verticalOffset;

  Address pattern2Address = pattern1Address + 8;
```

```
  Byte pattern1 = nes_readPPUMemory(nes, pattern1Address);
  Byte pattern2 = nes_readPPUMemory(nes, pattern2Address);

  Address patternIndex = ppu_combinePatternBytes(ppu, pattern1, pattern2, x);

  return patternIndex;
}

static Byte ppu_getBackgroundAttributeByte(NES nes) {
  assert(nes != NULL);
  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);


  Address nametableStart = ppu_getCurrentBaseNametableAddress(ppu);
  assert(nametableStart == PPU_NAME_TABLE_0_FIRST_ADDRESS ||
         nametableStart == PPU_NAME_TABLE_1_FIRST_ADDRESS ||
         nametableStart == PPU_NAME_TABLE_2_FIRST_ADDRESS ||
         nametableStart == PPU_NAME_TABLE_3_FIRST_ADDRESS);

  Word x = ppu_getCurrentX(ppu);
  Word y = ppu_getCurrentY(ppu);

  assert(PPU_NAMETABLE_BYTES_BEFORE_ATTRIBUTE_TABLE == 960);

  Address attributetableStart = nametableStart +
PPU_NAMETABLE_BYTES_BEFORE_ATTRIBUTE_TABLE;

  Word tileNumber = ppu_getBackgroundTileNumber(x, y);

  Word tileRowNumber = (tileNumber / PPU_BACKGROUND_TILES_PER_ROW);

  Word tileColumnNumber = tileNumber % PPU_BACKGROUND_TILES_PER_ROW;

  Word horizontalOffset = (tileColumnNumber / PPU_HORIZONTAL_TILES_PER_ATTRIBUTE_BYTE);

  Word verticalOffset = (tileRowNumber / PPU_VERTICAL_TILES_PER_ATTRIBUTE_BYTE) *
PPU_ATTRIBUTE_BYTES_PER_ROW;

  Address attributeByteAddress = attributetableStart + horizontalOffset +
verticalOffset;

  VALIDATE_NAMETABLE_ADDRESS(attributeByteAddress);

  Byte attributeByte = nes_readPPUMemory(nes,attributeByteAddress);
  return attributeByte;
}


static Byte ppu_getBackgroundAttributeColourIndex(NES nes) {
  assert(nes != NULL);
  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  Byte attributeByte = ppu_getBackgroundAttributeByte(nes);

  Word x = ppu_getCurrentX(ppu);
  Word y = ppu_getCurrentY(ppu);

  Word tileNumber = ppu_getBackgroundTileNumber(x, y);

  Word horizontalOffset = (tileNumber % PPU_HORIZONTAL_TILES_PER_ATTRIBUTE_BYTE);

  Word tileRowNumber = (tileNumber / PPU_BACKGROUND_TILES_PER_ROW);

  Word verticalOffset = (tileRowNumber % PPU_VERTICAL_TILES_PER_ATTRIBUTE_BYTE);

  int attributeTileNumber = attributeTableLookup[verticalOffset][horizontalOffset];

  Byte attributeColourIndex = 0;
```

```
  switch(attributeTileNumber) {

    case 0x0:
    case 0x1:
    case 0x2:
    case 0x3:
      attributeColourIndex = (attributeByte << 6 );
      break;

    case 0x4:
    case 0x5:
    case 0x6:
    case 0x7:
      attributeColourIndex = (attributeByte << 4 );
      break;

    case 0x8:
    case 0x9:
    case 0xA:
    case 0xB:
      attributeColourIndex = (attributeByte << 2 );
      break;

    case 0xC:
    case 0xD:
    case 0xE:
    case 0xF:
      attributeColourIndex = attributeByte;
      break;

  }

  attributeColourIndex = attributeColourIndex >> 6;
  attributeColourIndex = attributeColourIndex << 2;

  return attributeColourIndex;
}

static Byte ppu_getSystemIndexFromBackgroundIndex(NES nes, Byte backgroundColourIndex) {
  assert(nes != NULL);

  Address address = PPU_BACKGROUND_PALETTE_FIRST_ADDRESS;

  address += backgroundColourIndex;

  Byte systemIndex = nes_readPPUMemory(nes, address);

  return systemIndex;
}

static Byte ppu_getCurrentBackgroundColour(NES nes) {
  assert(nes != NULL);
  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  Address basePatternAddress;

  if (ppu_getControlBackgroundTileTable(ppu) == FALSE) {
    basePatternAddress = PPU_PATTERN_TABLE_0_FIRST_ADDRESS;
  } else {
    basePatternAddress = PPU_PATTERN_TABLE_1_FIRST_ADDRESS;
  }

  Byte x = ppu_getCurrentX(ppu);
  Byte y = ppu_getCurrentY(ppu);

  Byte patternTileNumber = ppu_getBackgroundNametableByte(nes);

  Byte patternColourIndex = ppu_getPatternColourIndex(nes, basePatternAddress,
patternTileNumber, x, y);

  Byte attributeColourIndex = ppu_getBackgroundAttributeColourIndex(nes);
```

```
  Byte backgroundColourIndex = 0;

  // if transparent, use the background colour
  if (patternColourIndex == 0) {
    backgroundColourIndex = 0;
  } else {
    backgroundColourIndex = patternColourIndex + attributeColourIndex ;
  }

  if (ppu_getMaskBackgroundVisibility(ppu) == FALSE) {
    backgroundColourIndex = 0;
  }

  return backgroundColourIndex;
}

static Byte ppu_getSystemIndexFromSpriteIndex(NES nes, Byte spriteColourIndex) {
  assert(nes != NULL);

  Address address = PPU_SPRITE_PALETTE_FIRST_ADDRESS;

  address += spriteColourIndex;

  Byte systemIndex = nes_readPPUMemory(nes, address);

  return systemIndex;
}


static void ppu_updateScanlineSpriteColour8(NES nes, Byte spriteIndex, Address
basePatternAddress, Byte multiplier) {
  assert(nes != NULL);
  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  Memory objectAttributeMemory = nes_getObjectAttributeMemory(nes);
  assert(objectAttributeMemory != NULL);

  Byte screenY = ppu_getCurrentY(ppu);

  Byte spriteLeftX = objectAttributeMemory_getX(objectAttributeMemory, spriteIndex);
  Byte spriteTopY = objectAttributeMemory_getY(objectAttributeMemory, spriteIndex) - 1;

  int yOffset = 0;
  Boolean inRange = FALSE;

  if (screenY >= spriteTopY) {
    yOffset = screenY - spriteTopY;
    if (yOffset < (PPU_VERTICAL_PIXELS_PER_TILE*multiplier)) {
      inRange = TRUE;
    }
  }

  Byte y = yOffset;

  Byte patternTileNumber = objectAttributeMemory_getTileNumber(objectAttributeMemory,
spriteIndex, FALSE);

  if (yOffset >= 8) {
    y = yOffset - 8;
    patternTileNumber++;
  }

  if (objectAttributeMemory_isFlippedVertical(objectAttributeMemory, spriteIndex) ==
TRUE) {
    y = PPU_VERTICAL_PIXELS_PER_TILE - y - 1;
  }

  Byte attributeIndex = objectAttributeMemory_getPalette(objectAttributeMemory,
spriteIndex);
```

```
  if (inRange == TRUE) {
    int i;
    for (i=0; i < PPU_HORIZONTAL_PIXELS_PER_TILE; i++) {
      if (ppu->spriteColoursForScanlineSet[spriteLeftX+i] == FALSE) {
        Byte x = i;

        if (objectAttributeMemory_isFlippedHorizontal(objectAttributeMemory,
spriteIndex) == TRUE) {
          x = PPU_HORIZONTAL_PIXELS_PER_TILE - x - 1;
        }

        Byte patternIndex = ppu_getPatternColourIndex(nes, basePatternAddress,
patternTileNumber, x, y);
        if (patternIndex != 0) {
          ppu->spriteColoursForScanline[spriteLeftX +i] = patternIndex + attributeIndex;
          ppu->spriteColoursForScanlineSet[spriteLeftX+i] = TRUE;
          ppu->spriteColoursForScanlineIsBehindBackground[spriteLeftX +i] =
objectAttributeMemory_isBehindBackground(objectAttributeMemory, spriteIndex);
        }
      }
    }
  }
}

// this is almost a copy paste of ppu_updateScanlineSpriteColour8
// think of a better way
static void ppu_updateScanlineSpriteColour16(NES nes, Byte spriteIndex) {
    assert(nes != NULL);
  Memory objectAttributeMemory = nes_getObjectAttributeMemory(nes);
  assert(objectAttributeMemory != NULL);

  Address basePatternAddress = objectAttributeMemory_getBankNumber(objectAttributeMemory,
spriteIndex);

  ppu_updateScanlineSpriteColour8(nes, spriteIndex, basePatternAddress, 2);
}


static void ppu_calculateSpriteColoursForCurrentScanline(NES nes) {
  assert(nes != NULL);
  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  ppu_resetSpriteColoursForScanline(ppu);

  if ( ppu_getMaskSpriteVisibility(ppu) == TRUE) {
    Byte spriteIndex;

    for (spriteIndex = 0; spriteIndex < OAM_NUMBER_OF_SPRITES; spriteIndex++) {

      Address basePatternAddress;

      if (ppu_getControlSpriteTileTable(ppu) == FALSE) {
        basePatternAddress = PPU_PATTERN_TABLE_0_FIRST_ADDRESS;
      } else {
        basePatternAddress = PPU_PATTERN_TABLE_1_FIRST_ADDRESS;
      }

      if (ppu_getControlSpriteSize(ppu) == TRUE) {
        ppu_updateScanlineSpriteColour16(nes, spriteIndex);
      } else {
        ppu_updateScanlineSpriteColour8(nes, spriteIndex, basePatternAddress, 1);
      }
    }
  }
}

static Byte ppu_getCurrentSpriteColour(NES nes, Byte currentBackgroundColour) {
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);
```

```
  Byte screenX = ppu_getCurrentX(ppu);

  Byte systemColourIndex = 0;

  if (ppu->spriteColoursForScanlineSet[screenX] == FALSE) {

    systemColourIndex =  ppu_getSystemIndexFromBackgroundIndex(nes,
currentBackgroundColour);

                                                                                //
one of the lower two bits set
  } else if ((ppu->spriteColoursForScanlineIsBehindBackground[screenX] == TRUE) &&
(currentBackgroundColour % 4 != 0)) {

    systemColourIndex = ppu_getSystemIndexFromBackgroundIndex(nes,
currentBackgroundColour);
    ppu_setStatusSpriteCollisionHit(ppu, TRUE);

  } else {

    ppu_setStatusSpriteCollisionHit(ppu, TRUE);
    systemColourIndex = ppu_getSystemIndexFromSpriteIndex(nes, ppu-
>spriteColoursForScanline[screenX]);

  }

  assert(systemColourIndex < PPU_NUM_SYSTEM_COLOURS);

  return systemColourIndex;
}

static void ppu_renderCurrentPixel(NES nes) {
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  GUI gui = nes_getGUI(nes);
  assert(gui != NULL);

  if (ppu->currentScanlineCycle == 0) {
    ppu_calculateSpriteColoursForCurrentScanline(nes);
  }

  Byte currentBackgroundColour = ppu_getCurrentBackgroundColour(nes);

  Byte currentSpriteColour = ppu_getCurrentSpriteColour(nes, currentBackgroundColour);

  Colour colour =
colour_init(systemPalette[currentSpriteColour].red,systemPalette[currentSpriteColour].gr
een,systemPalette[currentSpriteColour].blue);


gui_drawPixel(gui,ppu_getCurrentX(ppu),ppu_getCurrentY(ppu),colour_getRed(colour),colour
_getGreen(colour),colour_getBlue(colour));

  colour_destroy(colour);
}



void ppu_step(NES nes) {
  assert(nes != NULL);
  GUI gui = nes_getGUI(nes);
  assert(gui != NULL);

  debug_printf("ppu_step\n");

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);
```

```
  if (ppu->currentCycle == 0) {
    ppu->statusRegister = 0; // everything is cleared on line 0
  }

  if (ppu->currentScanline >=0 && ppu->currentScanline <= 19) {

    // do nothing, we are in vblank

  } else if (ppu->currentScanline == 20) {

    // debug_printf("%d\n", ppu->currentCycle);

    /*
    20: After 20 scanlines worth of time go by (since the VINT flag was set),
    the PPU starts to render scanlines. This first scanline is a dummy one;
    although it will access it's external memory in the same sequence it would
    for drawing a valid scanline, no on-screen pixels are rendered during this
    time, making the fetched background data immaterial. Both horizontal *and*
    vertical scroll counters are updated (presumably) at cc offset 256 in this
    scanline. Other than that, the operation of this scanline is identical to
    any other. The primary reason this scanline exists is to start the object
    render pipeline, since it takes 256 cc's worth of time to determine which
    objects are in range or not for any particular scanline.
    */

  } else if (ppu->currentScanline >= 21 && ppu->currentScanline <= 260) {

    /*
    21..260: after rendering 1 dummy scanline, the PPU starts to render the
    actual data to be displayed on the screen. This is done for 240 scanlines,
    of course.
    */

    if (ppu->currentScanlineCycle >=0 && ppu->currentScanlineCycle <
PPU_SCREEN_WIDTH_IN_PIXELS) {
      ppu_renderCurrentPixel(nes);
    }

  } else if (ppu->currentScanline == 261) {

    /*
    261:       after the very last rendered scanline finishes, the PPU does nothing
    for 1 scanline (i.e. the programmer gets screwed out of perfectly good VINT
    time). When this scanline finishes, the VINT flag is set, and the process of
    drawing lines starts all over again.
    */

  } else {

    assert(FALSE);
  }

  ppu->currentCycle++;
  ppu->currentScanlineCycle++;

  if (ppu->currentCycle % PPU_CYCLES_PER_SCANLINE == 0) {

    ppu->currentScanline++;

    ppu->currentScanlineCycle = 0;

    if (ppu->currentScanline == 20) {

      // The VBL flag is cleared 6820 PPU clocks, or exactly 20 scanlines, after it is
set.

      assert(ppu->currentCycle == 6820);

      ppu_setStatusSpriteOverflow(ppu, FALSE);

      ppu_setStatusSpriteCollisionHit(ppu, FALSE);
```

```
      ppu_setStatusVerticalBlank(ppu, FALSE);

    } else if (ppu->currentScanline % PPU_SCANLINES_PER_FRAME == 0) {

      // $2002.5 and $2002.6 after being set, stay that way for the first 20
      // scanlines of the new frame, relative to the VINT.

      ppu_setStatusVerticalBlank(ppu, TRUE);

      ppu->currentCycle = 0;
      ppu->currentScanline = 0;

      ppu->currentFrame++;

      gui_refresh(gui);

      printf("Frame: %d\n", ppu->currentFrame);

      if (ppu_getControlNMI(ppu) == TRUE) {
        nes_generateNMI(nes);
      }
    }
  }
}

Byte ppu_getControlRegister(PPU ppu) {
  assert(ppu != NULL);
  return ppu->controlRegister;
}

Byte ppu_getMaskRegister(PPU ppu) {
  assert(ppu != NULL);
  return ppu->maskRegister;
}

Byte ppu_getStatusRegister(PPU ppu) {
  assert(ppu != NULL);

  Byte status = ppu->statusRegister;

  // cleared on read
  ppu_setStatusVerticalBlank(ppu, FALSE);

  ppu->hasPartial = FALSE;

  return status;
}

Byte ppu_getSpriteAddressRegister(PPU ppu) {
  assert(ppu != NULL);
  return ppu->spriteAddressRegister;
}

Byte ppu_getSpriteDataRegister(NES nes) {
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  Byte data = nes_readObjectAttributeMemory(nes, ppu->spriteAddressRegister);

  //reads don't increase this?
  //ppu->spriteAddressRegister++;

  return data;
}

void ppu_setSpriteDataRegister(NES nes, Byte spriteDataRegister) {
  assert(nes != NULL);

  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);
```

```
  nes_writeObjectAttributeMemory(nes, ppu->spriteAddressRegister, spriteDataRegister);

  ppu->spriteAddressRegister++;
}

Byte ppu_getScrollRegister(PPU ppu) {
  assert(ppu != NULL);
  return ppu->scrollRegister;
}

Byte ppu_getPPUMemoryAddressRegister(PPU ppu) {
  assert(ppu != NULL);
  return ppu->ppuMemoryAddressRegister;
}


void ppu_setControlRegister(PPU ppu, Byte controlRegister) {
  assert(ppu != NULL);
  ppu->controlRegister = controlRegister;
}

void ppu_setMaskRegister(PPU ppu, Byte maskRegister) {
  assert(ppu != NULL);
  ppu->maskRegister = maskRegister;
}

void ppu_setStatusRegister(PPU ppu, Byte statusRegister) {
  assert(ppu != NULL);
  ppu->statusRegister = statusRegister;
}

void ppu_setSpriteAddressRegister(PPU ppu, Byte spriteAddressRegister) {
  assert(ppu != NULL);
  ppu->spriteAddressRegister = spriteAddressRegister;
}

void ppu_setScrollRegister(PPU ppu, Byte scrollRegister) {
  assert(ppu != NULL);

  if (ppu->hasPartial == FALSE) {

    ppu->horizontalScroll = scrollRegister;
    ppu->hasPartial = TRUE;

  } else {
    // we have a partial scroll

    ppu->verticalScroll = scrollRegister;
    ppu->hasPartial = FALSE;
  }

  ppu->scrollRegister = scrollRegister;
}

void ppu_setPPUMemoryAddressRegister(PPU ppu, Byte ppuMemoryAddressRegister) {
  assert(ppu != NULL);

  if (ppu->hasPartial == FALSE) {

    ppu->ppuAddressHigh = ppuMemoryAddressRegister;
    ppu->hasPartial = TRUE;

  } else {
    // we have a partial address

    ppu->ppuAddressLow = ppuMemoryAddressRegister;
    ppu->hasPartial = FALSE;
  }

  ppu->ppuMemoryAddressRegister = ppuMemoryAddressRegister;
}
```

```
static void ppu_increasePPUMemoryAddress(PPU ppu) {
  //assert(ppu->hasPartial == FALSE);

  Address address = ppu->ppuAddressLow;
  address += ppu->ppuAddressHigh << BITS_PER_BYTE;


  if (ppu_getControlPPUAddressIncrement(ppu) == TRUE) {
    address += PPU_CONTROL_ADDRESS_VERTICAL_INCREMENT;
  } else {
    address += PPU_CONTROL_ADDRESS_HORIZONTAL_INCREMENT;
  }


  ppu->ppuAddressLow = GET_ADDRESS_LOW_BYTE(address);
  ppu->ppuAddressHigh = GET_ADDRESS_HIGH_BYTE(address);
}

void ppu_setPPUMemoryDataRegister(NES nes, Byte ppuMemoryDataRegister) {
  assert(nes != NULL);
  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  Address address = ppu->ppuAddressLow;
  address += ppu->ppuAddressHigh << BITS_PER_BYTE;

  nes_writePPUMemory(nes, address, ppuMemoryDataRegister);

  //ppu->ppuMemoryDataRegister = ppuMemoryDataRegister;

  ppu_increasePPUMemoryAddress(ppu);
}

// one buffer pipeline
Byte ppu_getPPUMemoryDataRegister(NES nes) {
  assert(nes != NULL);
  PPU ppu = nes_getPPU(nes);
  assert(ppu != NULL);

  Byte data = ppu->ppuMemoryReadBuffer;

  Address address = ppu->ppuAddressLow;
  address += ppu->ppuAddressHigh << BITS_PER_BYTE;

  ppu->ppuMemoryReadBuffer = nes_readPPUMemory(nes, address);

  while (address > PPU_LAST_REAL_ADDRESS) {
    address -= PPU_NUM_REAL_ADDRESSES;
  }
  assert(address <= PPU_LAST_REAL_ADDRESS);

  // palette reads are not buffered
  if (address >= PPU_BACKGROUND_PALETTE_FIRST_ADDRESS) {
    data = ppu->ppuMemoryReadBuffer;
  }

  ppu_increasePPUMemoryAddress(ppu);

  return data;
}

void ppu_test() {

  {
    PPU ppu = ppu_init();
    assert(ppu != NULL);

    ppu_setStatusVerticalBlank(ppu, TRUE);
    assert(ppu_getStatusVerticalBlank(ppu) == TRUE);

    ppu_setStatusVerticalBlank(ppu, FALSE);
```

```
  assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
}

{

  PPU ppu = ppu_init();
  assert(ppu != NULL);


  ppu_setControlPPUAddressIncrement(ppu, TRUE);

  assert(ppu_getControlPPUAddressIncrement(ppu) == TRUE);
  assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
  assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
  assert(ppu_getControlSpriteSize(ppu) == FALSE);
  assert(ppu_getControlNMI(ppu) == FALSE);
  assert(ppu_getMaskDisplayType(ppu) == FALSE);
  assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
  assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
  assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
  assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

  assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
  assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

  assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


  ppu_setControlPPUAddressIncrement(ppu, FALSE);

  assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
  assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
  assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
  assert(ppu_getControlSpriteSize(ppu) == FALSE);
  assert(ppu_getControlNMI(ppu) == FALSE);

  assert(ppu_getMaskDisplayType(ppu) == FALSE);
  assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
  assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
  assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
  assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

  assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
  assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

  assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


  ppu_setControlSpriteTileTable(ppu, TRUE);

  assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
  assert(ppu_getControlSpriteTileTable(ppu) == TRUE);
  assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
  assert(ppu_getControlSpriteSize(ppu) == FALSE);
  assert(ppu_getControlNMI(ppu) == FALSE);

  assert(ppu_getMaskDisplayType(ppu) == FALSE);
  assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
  assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
  assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
  assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

  assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
  assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

  assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


  ppu_setControlSpriteTileTable(ppu, FALSE);

  assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
  assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
```

```
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


ppu_setControlBackgroundTileTable(ppu, TRUE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == TRUE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


ppu_setControlBackgroundTileTable(ppu, FALSE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


ppu_setControlSpriteSize(ppu, TRUE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == TRUE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);
```

```
assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


ppu_setControlSpriteSize(ppu, FALSE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);

ppu_setControlNMI(ppu, TRUE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == TRUE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


ppu_setControlNMI(ppu, FALSE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


ppu_setStatusVerticalBlank(ppu, TRUE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
```

```
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == TRUE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


ppu_setStatusVerticalBlank(ppu, FALSE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


ppu_setMaskDisplayType(ppu, TRUE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == TRUE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


ppu_setMaskDisplayType(ppu, FALSE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);
```

```
ppu_setMaskBackgroundClipping(ppu, TRUE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == TRUE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


ppu_setMaskBackgroundClipping(ppu, FALSE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);



ppu_setMaskSpriteClipping(ppu, TRUE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == TRUE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


ppu_setMaskSpriteClipping(ppu, FALSE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
```

```
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);



ppu_setMaskBackgroundVisiblity(ppu, TRUE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == TRUE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


ppu_setMaskBackgroundVisiblity(ppu, FALSE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);



ppu_setMaskSpriteVisibility(ppu, TRUE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == TRUE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);
```

```
ppu_setMaskSpriteVisibility(ppu, FALSE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);



ppu_setStatusSpriteOverflow(ppu, TRUE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == TRUE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


ppu_setStatusSpriteOverflow(ppu, FALSE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);



ppu_setStatusSpriteCollisionHit(ppu, TRUE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
```

```
  assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
  assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
  assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
  assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

  assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
  assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

  assert(ppu_getStatusSpriteCollisionHit(ppu) == TRUE);


  ppu_setStatusSpriteCollisionHit(ppu, FALSE);

  assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
  assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
  assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
  assert(ppu_getControlSpriteSize(ppu) == FALSE);
  assert(ppu_getControlNMI(ppu) == FALSE);

  assert(ppu_getMaskDisplayType(ppu) == FALSE);
  assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
  assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
  assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
  assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);

  assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
  assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

  assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);

}

{

  PPU ppu = ppu_init();
  assert(ppu != NULL);

  cpu_setMarkIntensifyRed(ppu, TRUE);

  assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
  assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
  assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
  assert(ppu_getControlSpriteSize(ppu) == FALSE);
  assert(ppu_getControlNMI(ppu) == FALSE);

  assert(ppu_getMaskDisplayType(ppu) == FALSE);
  assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
  assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
  assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
  assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);
  assert(cpu_getMaskIntensifyRed(ppu) == TRUE);
  assert(cpu_getMaskIntensifyGren(ppu) == FALSE);
  assert(cpu_getMaskIntensifyBlue(ppu) == FALSE);


  assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
  assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

  assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


  cpu_setMarkIntensifyRed(ppu, FALSE);

  assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
  assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
  assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
  assert(ppu_getControlSpriteSize(ppu) == FALSE);
  assert(ppu_getControlNMI(ppu) == FALSE);

  assert(ppu_getMaskDisplayType(ppu) == FALSE);
  assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
  assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
```

```
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);
assert(cpu_getMaskIntensifyRed(ppu) == FALSE);
assert(cpu_getMaskIntensifyGren(ppu) == FALSE);
assert(cpu_getMaskIntensifyBlue(ppu) == FALSE);


assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


cpu_setMarkIntensifyGreen(ppu, TRUE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);
assert(cpu_getMaskIntensifyRed(ppu) == FALSE);
assert(cpu_getMaskIntensifyGren(ppu) == TRUE);
assert(cpu_getMaskIntensifyBlue(ppu) == FALSE);


assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


cpu_setMarkIntensifyGreen(ppu, FALSE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);
assert(cpu_getMaskIntensifyRed(ppu) == FALSE);
assert(cpu_getMaskIntensifyGren(ppu) == FALSE);
assert(cpu_getMaskIntensifyBlue(ppu) == FALSE);


assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


cpu_setMarkIntensifyBlue(ppu, TRUE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
```

```
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);
assert(cpu_getMaskIntensifyRed(ppu) == FALSE);
assert(cpu_getMaskIntensifyGren(ppu) == FALSE);
assert(cpu_getMaskIntensifyBlue(ppu) == TRUE);


assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


cpu_setMarkIntensifyBlue(ppu, FALSE);

assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);
assert(cpu_getMaskIntensifyRed(ppu) == FALSE);
assert(cpu_getMaskIntensifyGren(ppu) == FALSE);
assert(cpu_getMaskIntensifyBlue(ppu) == FALSE);


assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


ppu_setControlHorizontalScrollNametable(ppu, TRUE);


assert(ppu_getControlHorizontalScrollNametable(ppu) == TRUE);
assert(ppu_getControlVerticalScrollNametable(ppu) == FALSE);
assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
assert(ppu_getControlSpriteSize(ppu) == FALSE);
assert(ppu_getControlNMI(ppu) == FALSE);

assert(ppu_getMaskDisplayType(ppu) == FALSE);
assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);
assert(cpu_getMaskIntensifyRed(ppu) == FALSE);
assert(cpu_getMaskIntensifyGren(ppu) == FALSE);
assert(cpu_getMaskIntensifyBlue(ppu) == FALSE);


assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


ppu_setControlHorizontalScrollNametable(ppu, FALSE);


assert(ppu_getControlHorizontalScrollNametable(ppu) == FALSE);
assert(ppu_getControlVerticalScrollNametable(ppu) == FALSE);
assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
```

```
    assert(ppu_getControlSpriteSize(ppu) == FALSE);
    assert(ppu_getControlNMI(ppu) == FALSE);

    assert(ppu_getMaskDisplayType(ppu) == FALSE);
    assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
    assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
    assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
    assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);
    assert(cpu_getMaskIntensifyRed(ppu) == FALSE);
    assert(cpu_getMaskIntensifyGren(ppu) == FALSE);
    assert(cpu_getMaskIntensifyBlue(ppu) == FALSE);


    assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
    assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

    assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


    ppu_setControlVerticalScrollNametable(ppu, TRUE);


    assert(ppu_getControlHorizontalScrollNametable(ppu) == FALSE);
    assert(ppu_getControlVerticalScrollNametable(ppu) == TRUE);
    assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
    assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
    assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
    assert(ppu_getControlSpriteSize(ppu) == FALSE);
    assert(ppu_getControlNMI(ppu) == FALSE);

    assert(ppu_getMaskDisplayType(ppu) == FALSE);
    assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
    assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
    assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
    assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);
    assert(cpu_getMaskIntensifyRed(ppu) == FALSE);
    assert(cpu_getMaskIntensifyGren(ppu) == FALSE);
    assert(cpu_getMaskIntensifyBlue(ppu) == FALSE);


    assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
    assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

    assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);


    ppu_setControlVerticalScrollNametable(ppu, FALSE);

    assert(ppu_getControlHorizontalScrollNametable(ppu) == FALSE);
    assert(ppu_getControlVerticalScrollNametable(ppu) == FALSE);
    assert(ppu_getControlPPUAddressIncrement(ppu) == FALSE);
    assert(ppu_getControlSpriteTileTable(ppu) == FALSE);
    assert(ppu_getControlBackgroundTileTable(ppu) == FALSE);
    assert(ppu_getControlSpriteSize(ppu) == FALSE);
    assert(ppu_getControlNMI(ppu) == FALSE);

    assert(ppu_getMaskDisplayType(ppu) == FALSE);
    assert(ppu_getMaskBackgroundClipping(ppu) == FALSE);
    assert(ppu_getMaskSpriteClipping(ppu) == FALSE);
    assert(ppu_getMaskBackgroundVisibility(ppu) == FALSE);
    assert(ppu_getMaskSpriteVisibility(ppu) == FALSE);
    assert(cpu_getMaskIntensifyRed(ppu) == FALSE);
    assert(cpu_getMaskIntensifyGren(ppu) == FALSE);
    assert(cpu_getMaskIntensifyBlue(ppu) == FALSE);

    assert(ppu_getStatusVerticalBlank(ppu) == FALSE);
    assert(ppu_getStatusSpriteOverflow(ppu) == FALSE);

    assert(ppu_getStatusSpriteCollisionHit(ppu) == FALSE);
  }

}
```

ppu.h

```
#include "ppu_type.h"
#include "nes_type.h"
#include "globals.h"

#define PPU_CONTROL_ADDRESS_VERTICAL_INCREMENT 32
#define PPU_CONTROL_ADDRESS_HORIZONTAL_INCREMENT 1

#define PPU_CYCLES_PER_SCANLINE 341
#define PPU_SCANLINES_PER_FRAME 262

#define PPU_NAMETABLE_BYTES_BEFORE_ATTRIBUTE_TABLE (PPU_BACKGROUND_TILES_PER_ROW *
PPU_BACKGROUND_TILES_PER_COLUMN)

#define PPU_HORIZONTAL_TILES_PER_ATTRIBUTE_BYTE 4
#define PPU_VERTICAL_TILES_PER_ATTRIBUTE_BYTE 4

#define PPU_ATTRIBUTE_BYTES_PER_ROW 8
#define PPU_ATTRIBUTE_BYTES_PER_COLUMN 8

#define PPU_TILES_PER_ATTRIBUTE_BYTE 16

#define PPU_TILES_PER_ATTRIBUTE_BYTE_ROW (PPU_TILES_PER_ATTIRBUTE_BYTE *
PPU_ATTRIBUTE_BYTES_PER_ROW)


#define PPU_ATTIRBUTE_TABLE_OFFSET (PPU_BACKGROUND_TILES_PER_ROW *
PPU_BACKGROUND_TILES_PER_COLUMN)

#define PPU_PATTERN_BYTES_PER_TILE 16

#define PPU_PATTERN_TILE_SECOND_BYTE_OFFSET 8


#define PPU_BACKGROUND_TILES_PER_ROW 32
#define PPU_BACKGROUND_TILES_PER_COLUMN 30


#define PPU_HORIZONTAL_PIXELS_PER_TILE 8
#define PPU_VERTICAL_PIXELS_PER_TILE 8

#define VALIDATE_NAMETABLE_ADDRESS(X) { assert(X >= PPU_NAME_TABLE_0_FIRST_ADDRESS);
assert(X <= PPU_NAME_TABLE_MIRROR_LAST_ADDRESS); }

#define PPU_WASTED_VBLANK_SCANLINES 20
#define PPU_WASTED_PREFETCH_SCANLINES 1


// Vertical blank (0: not in VBLANK; 1: in VBLANK)
#define MASK_STATUS_VBLANK_ON MASK_BIT7
#define MASK_STATUS_VBLANK_OFF (~MASK_BIT7)


//     6      a primary object pixel has collided with a playfield pixel in the last
frame
// bit6: sprite collision hit

// Set when a nonzero pixel of sprite 0 'hits'
// a nonzero background pixel.  Used for raster timing.

#define MASK_STATUS_SPRITE_COLLISION_HIT_ON MASK_BIT6
#define MASK_STATUS_SPRITE_COLLISION_HIT_OFF (~MASK_BIT6)

// 2   5      more than 8 objects on a single scanline have been detected in the last
frame
// bit5: sprite overflow

// Sprite overflow. The PPU can handle only eight sprites on one
// scanline and sets this bit if it starts dropping sprites.
// Normally, this triggers when there are 9 sprites on a scanline,
```

```
// but the actual behavior is significantly more complicated.

#define MASK_STATUS_SPRITE_OVERFLOW_ON MASK_BIT5
#define MASK_STATUS_SPRITE_OVERFLOW_OFF (~MASK_BIT5)

// bit4: sprite visibility
// Enable sprite rendering

#define MASK_MASK_SPRITE_VISIBILITY_ON MASK_BIT4
#define MASK_MASK_SPRITE_VISIBILITY_OFF (~MASK_BIT4)


// bit3: background visibility
// Enable background rendering

#define MASK_MASK_BACKGROUND_VISIBILITY_ON MASK_BIT3
#define MASK_MASK_BACKGROUND_VISIBILITY_OFF (~MASK_BIT3)


//     2      left side screen column (8 pixels wide) object clipping (when 0).
// bit2: sprite clipping
// Disable sprite clipping in leftmost 8 pixels of screen

#define MASK_MASK_SPRITE_CLIPPING_ON MASK_BIT2
#define MASK_MASK_SPRITE_CLIPPING_OFF (~MASK_BIT2)

// 1   left side screen column (8 pixels wide) playfield clipping (when 0).
// bit1: background clipping
// Disable background clipping in leftmost 8 pixels of screen

#define MASK_MASK_BACKGROUND_CLIPPING_ON MASK_BIT1
#define MASK_MASK_BACKGROUND_CLIPPING_OFF (~MASK_BIT1)


// bit0: display type

// Grayscale (0: normal color; 1: AND all palette entries
// with 0x30, effectively producing a monochrome display;
// note that colour emphasis STILL works when this is on!)

#define MASK_MASK_DISPLAY_TYPE_ON MASK_BIT0
#define MASK_MASK_DISPLAY_TYPE_OFF (~MASK_BIT0)


#define MASK_CONTROL_NMI_ON MASK_BIT7
#define MASK_CONTROL_NMI_OFF (~MASK_BIT7)


// Sprite size (0: 8x8; 1: 8x16)
//     5      8/16 scanline objects (0/1)
#define MASK_CONTROL_SPRITE_SIZE_ON MASK_BIT5
#define MASK_CONTROL_SPRITE_SIZE_OFF (~MASK_BIT5)


// Background pattern table address (0: $0000; 1: $1000)
#define MASK_CONTROL_BACKGROUND_TILE_TABLE_ON MASK_BIT4
#define MASK_CONTROL_BACKGROUND_TILE_TABLE_OFF (~MASK_BIT4)


// Sprite pattern table address for 8x8 sprites (0: $0000; 1: $1000)
//     3      object pattern table selection (if bit 5 = 0)
#define MASK_CONTROL_SPRITE_TILE_TABLE_ON MASK_BIT3
#define MASK_CONTROL_SPRITE_TILE_TABLE_OFF (~MASK_BIT3)



#define MASK_CONTROL_PPU_ADDRESS_INCREMENT_ON MASK_BIT2
#define MASK_CONTROL_PPU_ADDRESS_INCREMENT_OFF (~MASK_BIT2)

//     7      B (to be documented)
// bit7: colour emphasis?
```

```
#define MASK_MASK_INTENSIFY_BLUE_ON MASK_BIT7
#define MASK_MASK_INTENSIFY_BLUE_OFF (~MASK_BIT7)

//      6       G (to be documented)
// bit6: colour emphasis?

#define MASK_MASK_INTENSIFY_GREEN_ON MASK_BIT6
#define MASK_MASK_INTENSIFY_GREEN_OFF (~MASK_BIT6)

// MASK_MASK

//      5       R (to be documented)
// bit5: colour emphasis?

#define MASK_MASK_INTENSIFY_RED_ON MASK_BIT5
#define MASK_MASK_INTENSIFY_RED_OFF (~MASK_BIT5) // (MASK_ALL_ON ^ MASK_BIT5)

// MASK_CONTROL

//  (0 = $2000; 1 = $2400; 2 = $2800; 3 = $2C00)

//      1       Y scroll name table selection.
// bit1 base name table?

#define MASK_CONTROL_VERTICAL_SCROLL_NAME_TABLE_ON MASK_BIT1
#define MASK_CONTROL_VERTICAL_SCROLL_NAME_TABLE_OFF (~MASK_BIT1) // (MASK_ALL_ON ^
MASK_BIT1)

// bit0 base name table?
// 0    0       X scroll name table selection.

#define MASK_CONTROL_HORIZONTAL_SCROLL_NAME_TABLE_ON MASK_BIT0
#define MASK_CONTROL_HORIZONTAL_SCROLL_NAME_TABLE_OFF (~MASK_BIT0) // (MASK_ALL_ON ^
MASK_BIT0)

#define PPU_SCREEN_WIDTH_IN_PIXELS 256

#define PPU_NUM_SYSTEM_COLOURS 64


PPU ppu_init(void);

void ppu_destroy(PPU ppu);

void ppu_step(NES nes);

void ppu_test();

Byte ppu_getControlRegister(PPU ppu);

Byte ppu_getMaskRegister(PPU ppu);
Byte ppu_getStatusRegister(PPU ppu);
Byte ppu_getSpriteAddressRegister(PPU ppu);

Byte ppu_getScrollRegister(PPU ppu);
Byte ppu_getPPUMemoryAddressRegister(PPU ppu);

void ppu_setControlRegister(PPU ppu, Byte controlRegister);
void ppu_setMaskRegister(PPU ppu, Byte maskRegister);
void ppu_setStatusRegister(PPU ppu, Byte statusRegister);
void ppu_setSpriteAddressRegister(PPU ppu, Byte spriteAddressRegister);

void ppu_setScrollRegister(PPU ppu, Byte scrollRegister);
void ppu_setPPUMemoryAddressRegister(PPU ppu, Byte ppuMemoryAddressRegister);

void ppu_setPPUMemoryDataRegister(NES nes, Byte ppuMemoryDataRegister);
Byte ppu_getPPUMemoryDataRegister(NES nes);

Byte ppu_getSpriteDataRegister(NES nes);
void ppu_setSpriteDataRegister(NES nes, Byte spriteDataRegister);
```

ppu_type.h

```
#ifndef PPU_TYPE_H
  #define PPU_TYPE_H
  typedef struct ppu *PPU;
#endif
```

ppuMemory.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "memory.h"
#include "ppuMemory.h"
#include "nes.h"


////////////////////////////////////////////////////////////////////////////////
//

// Addresses $3F10/$3F14/$3F18/$3F1C are mirrors of $3F00/$3F04/$3F08/$3F0C.

static void ppuMemory_transparencyMirror_writer(NES nes, Address address, Byte data) {
  Memory ppuMemory = nes_getPPUMemory(nes);
  assert(ppuMemory != NULL);

  address -= PPU_BACKGROUND_PALETTE_TOTAL_SIZE;

  assert(address >= PPU_BACKGROUND_PALETTE_FIRST_ADDRESS);
  assert(address <= PPU_BACKGROUND_PALETTE_LAST_ADDRESS);

  memory_write_callback(nes, ppuMemory, address, data);
}

static Byte ppuMemory_transparencyMirror_reader(NES nes, Address address) {
  Memory ppuMemory = nes_getPPUMemory(nes);
  assert(ppuMemory != NULL);

  address -= PPU_BACKGROUND_PALETTE_TOTAL_SIZE;

  assert(address >= PPU_BACKGROUND_PALETTE_FIRST_ADDRESS);
  assert(address <= PPU_BACKGROUND_PALETTE_LAST_ADDRESS);

  Byte data = memory_read_callback(nes, ppuMemory, address);

  return data;
}


////////////////////////////////////////////////////////////////////////////////
//

static Address ppuMemory_paletteMirror_getLowestAddress(Address address) {

  while(address > PPU_GENUINE_PALETTE_LAST_ADDRESS) {
    address -= PPU_PALETTE_MIRRORED_SIZE;
  }

  assert(address <= PPU_GENUINE_PALETTE_LAST_ADDRESS);

  return address;
}

// Both palettes are also mirrored to $3F20-$3FFF.

static void ppuMemory_paletteMirror_writer(NES nes, Address address, Byte data) {
  Memory ppuMemory = nes_getPPUMemory(nes);
  assert(ppuMemory != NULL);

  address = ppuMemory_paletteMirror_getLowestAddress(address);

  memory_write_callback(nes, ppuMemory, address, data);
}

static Byte ppuMemory_paletteMirror_reader(NES nes, Address address) {
  Memory ppuMemory = nes_getPPUMemory(nes);
  assert(ppuMemory != NULL);
```

```
  address = ppuMemory_paletteMirror_getLowestAddress(address);

  Byte data = memory_read_callback(nes, ppuMemory, address);

  return data;
}

//////////////////////////////////////////////////////////////////////////////////
//

static void ppuMemory_nametableMirror_writer(NES nes, Address address, Byte data) {
  Memory ppuMemory = nes_getPPUMemory(nes);
  assert(ppuMemory != NULL);

  address -= PPU_NAME_TABLE_TOTAL_SIZE;

  assert(address >= PPU_NAME_TABLE_0_FIRST_ADDRESS);
  assert(address <= PPU_NAME_TABLE_3_FIRST_ADDRESS);

  memory_write_callback(nes, ppuMemory, address, data);
}

static Byte ppuMemory_nametableMirror_reader(NES nes, Address address) {
  Memory ppuMemory = nes_getPPUMemory(nes);
  assert(ppuMemory != NULL);

  address -= PPU_NAME_TABLE_TOTAL_SIZE;

  assert(address >= PPU_NAME_TABLE_0_FIRST_ADDRESS);
  assert(address <= PPU_NAME_TABLE_3_FIRST_ADDRESS);

  Byte data = memory_read_callback(nes, ppuMemory, address);

  return data;
}

//////////////////////////////////////////////////////////////////////////////////
//

Memory ppuMemory_init(void) {
  Memory memory = memory_init(PPU_NUM_REAL_ADDRESSES);
  assert(memory != NULL);

  Address address;
  // first setup the name table mirror

  for (address = PPU_NAME_TABLE_MIRROR_FIRST_ADDRESS; address <=
PPU_NAME_TABLE_MIRROR_FIRST_ADDRESS; address++) {
    memory_setWriteCallback(memory, address, &ppuMemory_nametableMirror_writer);
    memory_setReadCallback(memory, address, &ppuMemory_nametableMirror_reader);
  }

  // setup the background mirrors for the first byte in each sub palette of the sprite
palette

  memory_setWriteCallback(memory, PPU_SPRITE_PALETTE_0_FIRST_ADDRESS,
&ppuMemory_transparencyMirror_writer);
  memory_setReadCallback(memory, PPU_SPRITE_PALETTE_0_FIRST_ADDRESS,
&ppuMemory_transparencyMirror_reader);

  memory_setWriteCallback(memory, PPU_SPRITE_PALETTE_1_FIRST_ADDRESS,
&ppuMemory_transparencyMirror_writer);
  memory_setReadCallback(memory, PPU_SPRITE_PALETTE_1_FIRST_ADDRESS,
&ppuMemory_transparencyMirror_reader);

  memory_setWriteCallback(memory, PPU_SPRITE_PALETTE_2_FIRST_ADDRESS,
&ppuMemory_transparencyMirror_writer);
  memory_setReadCallback(memory, PPU_SPRITE_PALETTE_2_FIRST_ADDRESS,
&ppuMemory_transparencyMirror_reader);

  memory_setWriteCallback(memory, PPU_SPRITE_PALETTE_3_FIRST_ADDRESS,
&ppuMemory_transparencyMirror_writer);
```

```
  memory_setReadCallback(memory, PPU_SPRITE_PALETTE_3_FIRST_ADDRESS,
&ppuMemory_transparencyMirror_reader);


  // now setup the mirror proper

  for (address=PPU_PALETTE_MIRROR_FIRST_ADDRESS; address <=
PPU_PALETTE_MIRROR_LAST_ADDRESS; address++) {
    memory_setWriteCallback(memory, address, &ppuMemory_paletteMirror_writer);
    memory_setReadCallback(memory, address, &ppuMemory_paletteMirror_reader);
  }

  return memory;
}
```

ppuMemory.h

```
#include "memory_type.h"

Memory ppuMemory_init(void);

#define PPU_FIRST_REAL_ADDRESS 0x0000
#define PPU_LAST_REAL_ADDRESS 0x3FFF

#define PPU_NUM_REAL_ADDRESSES (PPU_LAST_REAL_ADDRESS + 1)


/*
        |    $0000      |    $0FFF      |    $1000      | Tile Set #0           |
        |    $1000      |    $1FFF      |    $1000      | Tile Set #1           |
        +---------------+---------------+---------------+-----------------------+
        |    $2000      |    $23FF      |    $0400      | Name Table #0         |
        |    $2400      |    $27FF      |    $0400      | Name Table #1         |
        |    $2800      |    $2BFF      |    $0400      | Name Table #2         |
        |    $2C00      |    $2FFF      |    $0400      | Name Table #3         |
        +---------------+---------------+---------------+-----------------------+
        |    $3000      |    $3EFF      |    $3EFF      | Name Table Mirror *1  |
        |    $3F00      |    $3FFF      |    $0020      | Palette *2            |
        |    $4000      |    $FFFF      |    $C000      | Mirrors of Above *3   |
*/


// tile set 0
#define PPU_PATTERN_TABLE_0_FIRST_ADDRESS 0x0000
#define PPU_PATTERN_TABLE_0_LAST_ADDRESS  0x0FFF

// tile set 1
#define PPU_PATTERN_TABLE_1_FIRST_ADDRESS 0x1000
#define PPU_PATTERN_TABLE_1_LAST_ADDRESS  0x1FFF


#define PPU_NAME_TABLE_0_FIRST_ADDRESS 0x2000
#define PPU_NAME_TABLE_0_LAST_ADDRESS  0x23FF

#define PPU_NAME_TABLE_1_FIRST_ADDRESS 0x2400
#define PPU_NAME_TABLE_1_LAST_ADDRESS  0x27FF

#define PPU_NAME_TABLE_2_FIRST_ADDRESS 0x2800
#define PPU_NAME_TABLE_2_LAST_ADDRESS  0x2BFF

#define PPU_NAME_TABLE_3_FIRST_ADDRESS 0x2C00
#define PPU_NAME_TABLE_3_LAST_ADDRESS  0x2FFF

#define PPU_NAME_TABLE_SIZE (PPU_NAME_TABLE_0_LAST_ADDRESS -
PPU_NAME_TABLE_0_FIRST_ADDRESS + 1)

#define PPU_NAME_TABLE_TOTAL_SIZE (PPU_NAME_TABLE_3_LAST_ADDRESS -
PPU_NAME_TABLE_0_FIRST_ADDRESS + 1)

#define PPU_NAME_TABLE_MIRROR_FIRST_ADDRESS 0x3000
#define PPU_NAME_TABLE_MIRROR_LAST_ADDRESS 0x3EFF


/*
The palette for the background runs from VRAM $3F00 to $3F0F;

$3F00   Universal background color

$3F01-$3F03    Background palette 0

$3F05-$3F07    Background palette 1

$3F09-$3F0B    Background palette 2

$3F0D-$3F0F    Background palette 3
```

```
Addresses $3F04/$3F08/$3F0C can contain unique data,
though these values are not used by the PPU when rendering.
*/


#define PPU_BACKGROUND_PALETTE_FIRST_ADDRESS   0x3F00
#define PPU_BACKGROUND_PALETTE_LAST_ADDRESS    0x3F0F


#define PPU_BACKGROUND_PALETTE_TOTAL_SIZE (PPU_BACKGROUND_PALETTE_LAST_ADDRESS -
PPU_BACKGROUND_PALETTE_FIRST_ADDRESS + 1)


#define PPU_BACKGROUND_PALLETE_0_FIRST_ADDRESS 0x3F00
#define PPU_BACKGROUND_PALETTE_0_LAST_ADDRESS  0x3F03

#define PPU_BACKGROUND_PALETTE_1_FIRST_ADDRESS 0x3F04
#define PPU_BACKGROUND_PALETTE_1_LAST_ADDRESS  0x3F07

#define PPU_BACKGROUND_PALETTE_2_FIRST_ADDRESS 0x3F08
#define PPU_BACKGROUND_PALETTE_2_LAST_ADDRESS  0x3F0B

#define PPU_BACKGROUND_PALETTE_3_FIRST_ADDRESS 0x3F0C
#define PPU_BACKGROUND_PALETTE_3_LAST_ADDRESS  0x3F0F


/*

the palette for the sprites runs from $3F10 to $3F1F.

$3F11-$3F13    Sprite palette 0

$3F15-$3F17    Sprite palette 1

$3F19-$3F1B    Sprite palette 2

$3F1D-$3F1F    Sprite palette 3

*/


#define PPU_SPRITE_PALETTE_FIRST_ADDRESS 0x3F10
#define PPU_SPRITE_PALETTE_LAST_ADDRESS  0x3F1F

#define PPU_SPRITE_PALETTE_0_FIRST_ADDRESS 0x3F10
#define PPU_SPRITE_PALETTE_0_LAST_ADDRESS  0x3F13

#define PPU_SPRITE_PALETTE_1_FIRST_ADDRESS 0x3F14
#define PPU_SPRITE_PALETTE_1_LAST_ADDRESS  0x3F17

#define PPU_SPRITE_PALETTE_2_FIRST_ADDRESS 0x3F18
#define PPU_SPRITE_PALETTE_2_LAST_ADDRESS  0x3F1B

#define PPU_SPRITE_PALETTE_3_FIRST_ADDRESS 0x3F1C
#define PPU_SPRITE_PALETTE_3_LAST_ADDRESS  0x3F1F

#define PPU_GENUINE_PALETTE_LAST_ADDRESS PPU_SPRITE_PALETTE_3_LAST_ADDRESS

#define PPU_PALETTE_MIRROR_FIRST_ADDRESS   (PPU_GENUINE_PALETTE_LAST_ADDRESS + 1) //
0x3F20
#define PPU_PALETTE_MIRROR_LAST_ADDRESS  0x3FFF


#define PPU_SPRITE_PALETTE_TOTAL_SIZE (PPU_SPRITE_PALETTE_LAST_ADDRESS -
PPU_SPRITE_PALETTE_FIRST_ADDRESS + 1)

#define PPU_PALETTE_MIRRORED_SIZE (PPU_BACKGROUND_PALETTE_TOTAL_SIZE +
PPU_SPRITE_PALETTE_TOTAL_SIZE)
```