codeslinger.co.uk

codeslinger.co.uk

Gameboy - LCD.

Scanlines:

The screen resolution is 160x144 meaning there are 144 visible scanlines. The Gameboy draws each scanline one at a time starting from 0 to 153, this means there are 144 visible scanlines and 8 invisible scanlines. When the current scanline is between 144 and 153 this is the vertical blank period. The current scanline is stored in register address 0xFF44. The pandocs tell us that it takes 456 cpu clock cycles to draw one scanline and move onto the next, so we will need a counter to know when to move onto the next line, we'll call this m_ScanlineCounter. Just like the timer and divider registers we can control the scanline counter by subtracting its value by the amount of clock cycles the last opcode took to exectue. If we look at our main emulator update loop that we emulated in the Getting Started chapter you can see that the UpdateGraphics function gets passed the amount of cycles the last opcode took so we can accurately keep the graphics in sync with the cpu. This is how the UpdateGraphics function is emulated:

```
void Emulator::UpdateGraphics(int cycles)
{
  SetLCDStatus( ) ;

  if (IsLCDEnabled())
    m_ScalineCounter -= cycles ;
  else
    return ;

  if (m_ScalineCounter <= 0)
  {
    // time to move onto next scanline
    m_Rom[0xFF44]++;
    BYTE currentline = ReadMemory(0xFF44) ;

    m_ScalineCounter = 456 ;

    // we have entered vertical blank period
    if (currentline == 144)
      RequestInterupt(0) ;

    // if gone past scanline 153 reset to 0
    else if (currentline > 153)
      m_Rom[0xFF44]=0

    // draw the current scanline
```

```
    else if (currentline < 144)
      DrawScanLine() ;
  }
}
```

There is quite a bit there that I will give implementations to later so for now I'll just explain it. If the lcd display is enabled then we need to update the scanline counter by subtracting the last opcodes cycle time from it. If this makes m_ScanlineCounter go negative then it is time to move onto the next scanline. Register 0xFF44 is the current scanline so we increment this onto then next scanline. If the new scanline is 144 then this is the beginning of the vetical blank interupt so we need to request this interupt (remember that the vblank interupt is bit 0). If you are unsure why this is then re-read the interupts chapter. If the current scanline is greater than 153 then it must be reset to 0. However if the current scanline is less than 144 then it is one of the visible scanlines and needs to be rendered to the screen. This will be explained in the chapter called Graphic Emulation. The reason why I access memory directly when writing to the current scanline address (0xFF44) rather than using WriteMemory is because when the game tries to write to 0xFF44 it resets the current scaline to 0, so we cannot use WriteMemory to increment the scanline as it will always set it to 0. Speaking of which the following needs to be added to the WriteMemory function:

```
// reset the current scanline if the game tries to write to it
else if (address == 0xFF44)
{
  m_Rom[address] = 0 ;
}
```

Setting the LCD Status:

The memory address 0xFF41 holds the current status of the LCD. The LCD goes through 4 different modes. These are "V-Blank Period", "H-Blank Period", "Searching Sprite Attributes" and "Transferring Data to LCD Driver". Bit 1 and 0 of the lcd status at address 0xFF41 reflects the current LCD mode like so:

00: H-Blank
01: V-Blank
10: Searching Sprites Atts
11: Transfering Data to LCD Driver

When starting a new scanline the lcd status is set to 2, it then moves on to 3 and then to 0. It then goes back to and continues then pattern until the v-blank period starts where it stays on mode 1. When the vblank period ends it goes back to 2 and continues this pattern over and over. As previously mentioned it takes 456 clock cycles to draw one scanline before moving onto the next. This can be split down into different sections which will represent the different modes. Mode 2 (Searching Sprites Atts) will take the first 80 of the 456 clock cycles. Mode 3 (Transfering to LCD Driver) will take 172 clock cycles of the 456 and the remaining clock cycles of the 456 is for Mode 0 (H-Blank). If this is confusing then please bare with me as it will all be come clear when you see the code.

When the LCD status changes its mode to either Mode 0, 1 or 2 then this can cause an LCD Interupt Request to happen. Bits 3, 4 and 5 of the LCD Status register (0xFF41) are interupt enabled flags (the same as the Interupt Enabled Register 0xFFFF, see interupts chapter). These bits are set by the

game not the emulator and they represent the following:

Bit 3: Mode 0 Interupt Enabled
Bit 4: Mode 1 Interupt Enabled
Bit 5: Mode 2 Interupt Enabled

So when the mode changes to 0,1 or 2 then if the corresponding bit 3,4,5 is set then an LCD interupt is requested. This is only tested when the LCD mode changes to 0,1 or 2 and not the duration of these modes.

One important part to emulate with the lcd modes is when the lcd is disabled the mode must be set to mode 1. If you dont do this then you will spend hours like I did wondering why Mario2 wont play past the title screen. You also need to reset the m_ScanlineCounter and current scanline

The last part of the LCD status register (0xFF41) is the Coincidence flag. Basically Bit 2 of the status register is set to 1 if register (0xFF44) is the same value as (0xFF45) otherwise it is set to 0. Bit 6 of the LCD status register (0xFF44) is the same as the interupt enabled bits 3-5 but it isnt to do with the current lcd mode it is to do with the bit 2 coincidence flag. If the conicidence flag (bit 2) is set and the conincidence interupt enabled flag (bit 6) is set then an LCD Interupt is requested. The conicidence flag means the current scanline (0xFF44) is the same as a scanline the game is interested in (0xFF45). The reason why the game would be interested in the current scanline is to do special effects. So when 0xFF44 == 0xFF45 then an interupt can be requested to let the game know that the values are the same.

We now have enough information to put all this together and implement the SetLCDStatus function

```
void Emulator::SetLCDStatus( )
{
  BYTE status = ReadMemory(0xFF41) ;
  if (false == IsLCDEnabled())
  {
    // set the mode to 1 during lcd disabled and reset scanline
    m_ScanlineCounter = 456 ;
    m_Rom[0xFF44] = 0 ;
    status &= 252 ;
    status = BitSet(status, 0) ;
    WriteMemory(0xFF41,status) ;
    return ;
  }

  BYTE currentline = ReadMemory(0xFF44) ;
  BYTE currentmode = status & 0x3 ;

  BYTE mode = 0 ;
  bool reqInt = false ;

  // in vblank so set mode to 1
  if (currentline >= 144)
  {     mode = 1;
    status = BitSet(status,0) ;
    status = BitReset(status,1) ;
```

```
      reqInt = TestBit(status,4) ;
    }
    else
    {
      int mode2bounds = 456-80 ;
      int mode3bounds = mode2bounds - 172 ;

      // mode 2
      if (m_ScanlineCounter >= mode2bounds)
      {
        mode = 2 ;
        status = BitSet(status,1) ;
        status = BitReset(status,0) ;
        reqInt = TestBit(status,5) ;
      }
      // mode 3
      else if(m_ScanlineCounter >= mode3bounds)
      {
        mode = 3 ;
        status = BitSet(status,1) ;
        status = BitSet(status,0) ;
      }
      // mode 0
      else
      {
        mode = 0;
        status = BitReset(status,1) ;
        status = BitReset(status,0) ;
        reqInt = TestBit(status,3) ;
      }
    }

    // just entered a new mode so request interupt
    if (reqInt && (mode != currentmode))
      RequestInterupt(1) ;

    // check the conincidence flag
    if (ly == ReadMemory(0xFF45))
    {
      status = BitSet(status,2) ;
      if (TestBit(status,6))
        RequestInterupt(1) ;
    }
    else
    {
      status = BitReset(status,2) ;
    }
    WriteMemory(0xFF41,status) ;
  }
```

Hopefully that all makes sense. If not re-read this section until it does. It took awhile for me to realise how the lcd status works

The LCD Control:

The LCD Control register (0xFF40) will be covered in detail in the chapter called Graphic Emulation however the above code uses the function IsLCDEnabled() a lot which I've yet to implement and this relies on the LCD Control register. Bit 7 of this register is responsible for enabling and disbaling the lcd so we can implement IsLCDEnabled() like so:

```
bool Emulator::IsLCDEnabled() const
{
  return TestBit(ReadMemory(0xFF40),7) ;
}
```

Thats it for the LCD chapter onto DMA Transfer