# Understanding HDMI

HDMI, based on the DVI standard, is made up of a bunch of different protocols. They can be divided as following;

- Low speed protocols
    - I2C - EDID/DDC
        - 
    - CEC
- High speed protocols
    - TMDS
        - Control Data - 10b2b
        - Pixel Data - 10b8b
        - Auxiliary Data - 10b4b
        - ***HDCP***
    - Ethernet

# Understanding TMDS 8b/10b encoding

The DVI and HDMI standards transmit video data using a standard called **T**ransition-**M**inimized **D**ifferential **S**ignaling (TMDS). One of the most important aspect of this standard is that it encodes the byte (8 bits) of each color channel data (red, green, blue) into 10 bits and then transmits the information serially at 10x the pixel clock.

The fact that TMDS converts the 8 bits per byte into 10 bits for transmission means the encoding format is frequently called "8b/10b". However, using this terminology is quite confusing as 8b/10b also refers to a totally different encoding scheme IBM developed that is used in many other protocols such as PCI Express and DisplayPort! To reiterate, the 8b/10b encoding scheme used in DVI (and thus also HDMI) is ***totally different*** to the IBM standard[1].

An important difference is that while both the TMDS and IBM schemes try to keep DC balance, TMDS sacrifices short term DC balance to reduce the interference between the 3 channels (RGB) that are sent side-by-side.

The TMDS encoding scheme is actually very simple but for some unknown reason described by this almost incomprehensible diagram found in the DVI specification:

[1] The IBM standard is a neat combination of 5b/6b and 3b/4b coding methods. The Wikipedia page is actually pretty readable description of how it works.

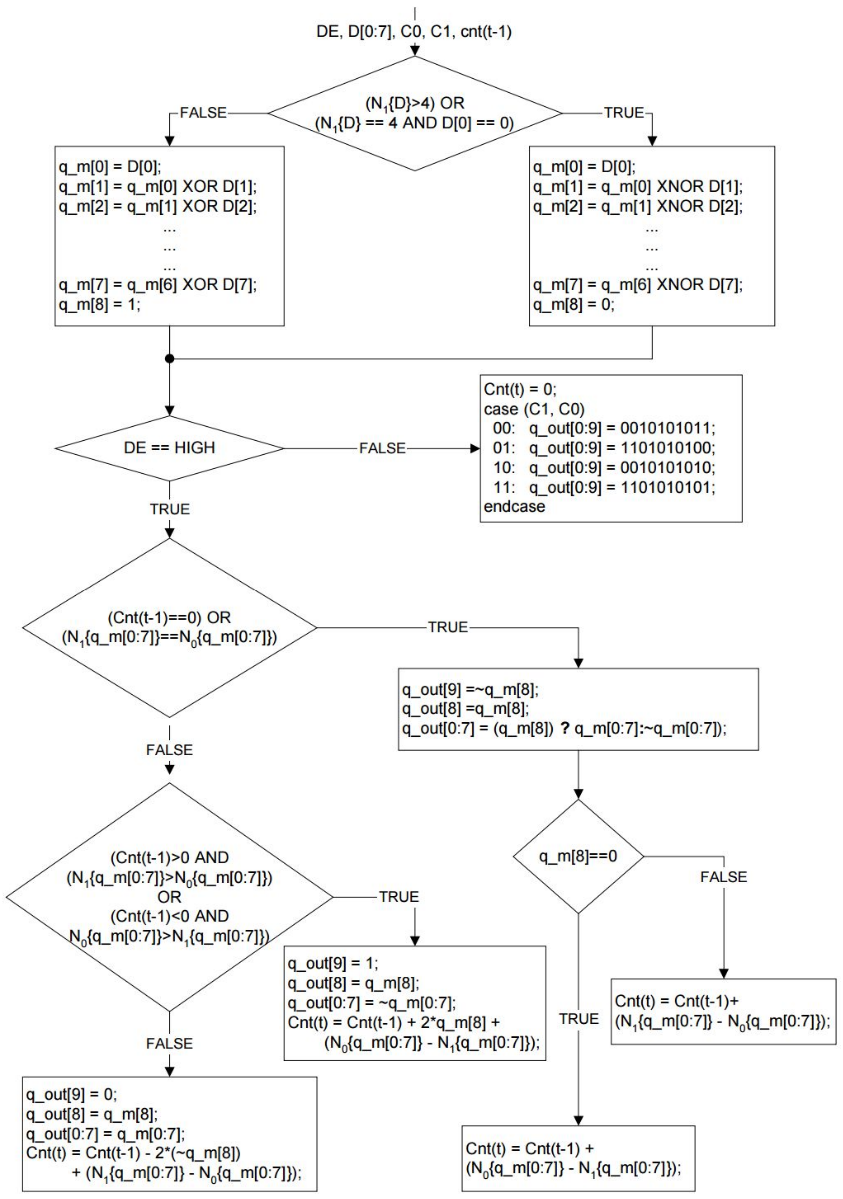

DE, D[0:7], C0, C1, cnt(t-1)
(N_1{D}>4) OR (N_1{D} == 4 AND D[0] == 0)
FALSE
TRUE
q_m[0] = D[0]; q_m[1] = q_m[0] XOR D[1]; q_m[2] = q_m[1] XOR D[2]; ... q_m[7] = q_m[6] XOR D[7]; q_m[8] = 1;
q_m[0] = D[0]; q_m[1] = q_m[0] XNOR D[1]; q_m[2] = q_m[1] XNOR D[2]; ... q_m[7] = q_m[6] XNOR D[7]; q_m[8] = 0;
DE == HIGH
FALSE
Cnt(t) = 0; case (C1, C0) 00: q_out[0:9] = 0010101011; 01: q_out[0:9] = 1101010100; 10: q_out[0:9] = 0010101010; 11: q_out[0:9] = 1101010101; endcase
TRUE
(Cnt(t-1)==0) OR (N_1{q_m[0:7]}==N_0{q_m[0:7]})
TRUE
q_out[9] =~q_m[8]; q_out[8] =q_m[8]; q_out[0:7] = (q_m[8]) ? q_m[0:7]:~q_m[0:7]);
FALSE
(Cnt(t-1)>0 AND (N_1{q_m[0:7]}>N_0{q_m[0:7]}) OR (Cnt(t-1)<0 AND N_0{q_m[0:7]}>N_1{q_m[0:7]})
TRUE
q_out[9] = 1; q_out[8] = q_m[8]; q_out[0:7] = ~q_m[0:7]; Cnt(t) = Cnt(t-1) + 2*q_m[8] + (N_0{q_m[0:7]} - N_1{q_m[0:7]});
FALSE
q_out[9] = 0; q_out[8] = q_m[8]; q_out[0:7] = q_m[0:7]; Cnt(t) = Cnt(t-1) - 2*(~q_m[8]) + (N_1{q_m[0:7]} - N_0{q_m[0:7]});
q_m[8]==0
FALSE
Cnt(t) = Cnt(t-1)+ (N_1{q_m[0:7]} - N_0{q_m[0:7]});
TRUE
Cnt(t) = Cnt(t-1) + (N_0{q_m[0:7]} - N_1{q_m[0:7]});

*Figure 3-5. T.M.D.S. Encode Algorithm*

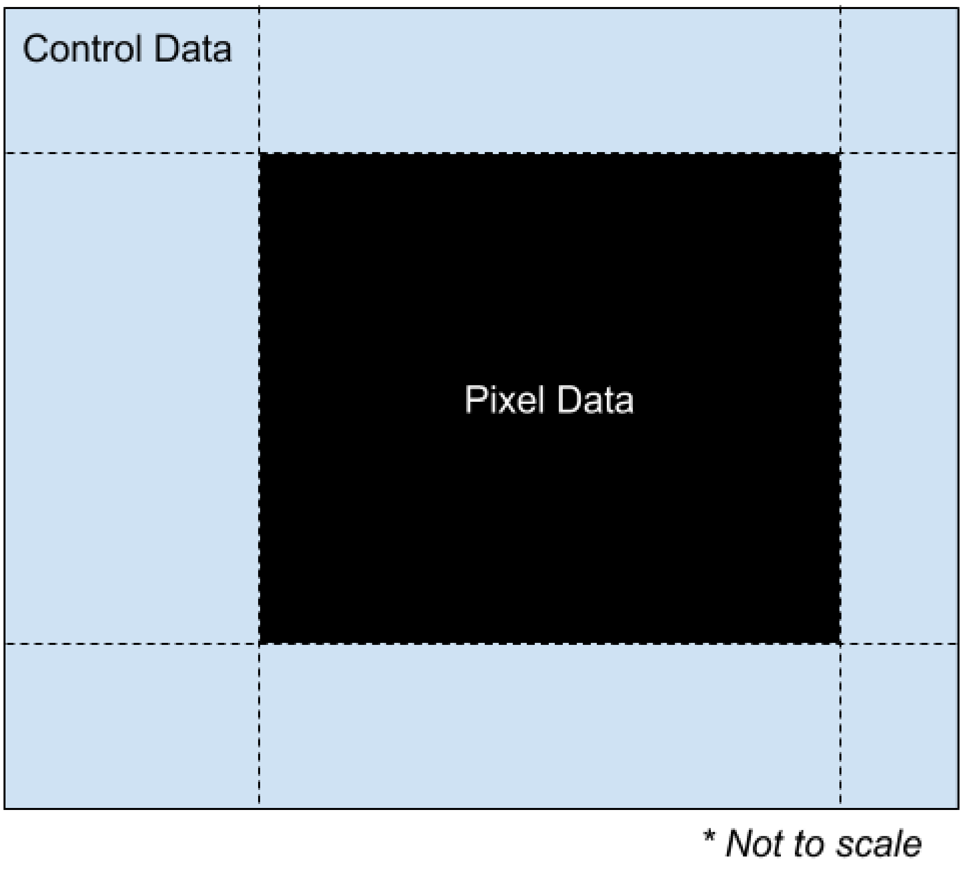
## Stage 0 - Pixel or Control Data

TMDS also uses almost two totally different encoding schemes depending on if the pixel data or control data is being transmitted.

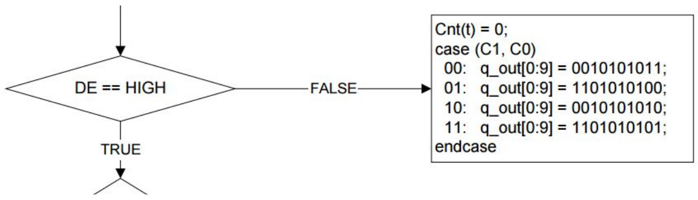
- Pixel data has 4 or fewer transitions in the first 8 bits.
- Control data has 6 or more transitions in the first 8 bits.

The pixel data contains the actual information which is displayed on the screen. The control data is mainly used for synchronization and transmitted during the "blanking periods" of the video signal[2] where no pixel data is available. For DVI/HDMI this is You can see how this would look in the following diagram;

[2] Blanking periods are left over from old CRT monitors needing to reset the position of the electron beam.



This is what the following part of the TMDS diagram is trying to explain[3]. The "DE" stands for "Data Enable" and is signalling that we are sending pixel data.



```
Cnt(t) = 0;
case (C1, C0)
  00:  q_out[0:9] = 0010101011;
  01:  q_out[0:9] = 1101010100;
  10:  q_out[0:9] = 0010101010;
  11:  q_out[0:9] = 1101010101;
endcase
```

Some resources for full details of VGA timing are;

- OS Dev Wiki - Video Signals and Timing
-

## Control Tokens

There are 4 fixed 10 bit control tokens which are used to transmit two bits of data called c0 and c1 (it could be called a 10b/2b encoding). These signals are mapped to the VSync and HSync video signals.

| | **Signals** | |
|---|---|---|
| **Channel** | **C0** | **C1** |
| 0 - Blue | HSYNC | VSYNC |
| 1 - Green | CTL0 | CTL1 |
| 2 - Red | CTL2 | CTL3 |

[3] Why this is put in the *middle* of the diagram, rather than as the first decision in the tree is unknown?

The control tokens are designed to be DC balanced.

| Data bits | | | Encoded bits | | | | | | | | | | DC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| c0 | c1 | Notes | A | B | C | D | E | F | G | H | X | I | Bias |
| 0 | 0 | Default token to be sent | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | +2 |
| 1 | 0 | | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | -2 |

```
yield ControlToken([0,0,1,0,1,0,1,0,1,1], c0=0, c1=0)
yield ControlToken([0,0,1,0,1,0,1,0,1,0], c0=0, c1=1)
yield ControlToken([1,1,0,1,0,1,0,1,0,0], c0=1, c1=0)
yield ControlToken([1,1,0,1,0,1,0,1,0,1], c0=1, c1=1)
```

The DVI protocol guarantees that you will get XXX control tokens every YYY pixel tokens. This allows receivers to use the control tokens to synchronize / deskew each of the pixel channels with the pixel clock.

## Data Island

One way the HDMI standard extends the DVI standard is by allowing yet another type of data tokens be sent. These are sent during long periods that would have been control tokens. They use yet another encoding scheme called TERC4 which allows 4 bits of data per channel be sent (hence it could be called a 10b/4b encoding scheme).

```
case (D3, D2, D1, D0):
0000: q_out[9:0] = 0b1010011100;
0001: q_out[9:0] = 0b1001100011;
0010: q_out[9:0] = 0b1011100100;
0011: q_out[9:0] = 0b1011100010;
0100: q_out[9:0] = 0b0101110001;
0101: q_out[9:0] = 0b0100011110;
0110: q_out[9:0] = 0b0110001110;
0111: q_out[9:0] = 0b0100111100;
1000: q_out[9:0] = 0b1011001100;
1001: q_out[9:0] = 0b0100111001;
1010: q_out[9:0] = 0b0110011100;
1011: q_out[9:0] = 0b1011000110;
1100: q_out[9:0] = 0b1010001110;
1101: q_out[9:0] = 0b1001110001;
1110: q_out[9:0] = 0b0101100011;
1111: q_out[9:0] = 0b1011000011;
endcase;
```

## Stage 1 - Pixel Data Transition Reduction Encoding

The first thing TMDS does is reduce the number of transitions in the data byte[4]. It does this by choosing between either an XOR or XNOR encoding method.

### XOR Encoding

Encoded Bit 0 == Data Bit 0
Encoded Bit 1 == Data Bit 1 XOR Encoded Bit 0
Encoded Bit 2 == Data Bit 2 XOR Encoded Bit 1
Encoded Bit 3 == Data Bit 3 XOR Encoded Bit 2
…

### XNOR Encoding

Encoded Bit 0 == Data Bit 0
Encoded Bit 1 == Data Bit 1 XNOR Encoded Bit 0
Encoded Bit 2 == Data Bit 2 XNOR Encoded Bit 1
Encoded Bit 3 == Data Bit 3 XNOR Encoded Bit 2
….

### Choosing the Encoding

The encoding method is determined by the number of "ones" (bits set) in the data byte;

- If fewer than 4 ones, use XOR
- If more than 4 ones, use XNOR
- If exactly 4 ones,
    - If data bit 0 is 1, use XOR
    - If data bit 0 is 0, use XNOR

The encoding method is entirely determined by the incoming data byte.

A 9th bit is added which describes which encoding method was used. "1" is added if the XOR scheme was used and a "0" is added if the XNOR scheme as used.

### Example Pixel Data Encoding

| | **Data bits** | | | | | | | | **Number of "Ones" in Data** | | **Encoded bits** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Data** | **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | | **Chosen Encoding** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **X** |
| 0x01 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | **XOR** -- ones<4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0x0F | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 4 | **XOR** -- ones==4 && data[0]==1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0x1E | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 4 | **XNOR** -- ones==4 && data[0]==0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0x1F | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 5 | **XNOR** -- ones>4 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |

[4] This reduction in transitions is probably why HDMI and DVI need to provide a dedicated pixel clock rather than allowing the clock to be recovered from the data transmission.

## Stage 2 - Inverting to keep DC balance

The encoding scheme however doesn't guarantee that the symbols have an even number of ones and zeros. This means that the symbols will cause a DC bias.

To solve that problem, TMDS will sometimes choose to invert a symbol. This then requires another bit to determine if the value was inverted.

Some of the symbols are balanced. For these symbols, we never invert them.

For some reason they don't invert the XOR/XNOR bit?

This means that the full symbol can be thought of as;

| **10 bit Symbol** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Encoded Data 8 bits** | | | | | | | | **Encoding format 1 bit** | **Inverted Symbol 1 bit** |
| A | B | C | D | E | F | G | H | X | I |

| **Encoded 9 bits** | | | | | | | | | | | | | | | | | | | | **DC Bias** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **X** | | | | | | | | | | | | |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | | | | | | | +8 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | -6 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | | | | | | | | | | | | | |

# VGA Signaling

http://labs.domipheus.com/blog/wp-content/uploads/2016/04/vga_800x600x60.png

```
    // $ xrandr --newmode "1280x1024_60.00"  109.00  1280 1368 1496 1712  1024 1027 1034 1063 -hsync +vsync
    // Based on code from http://cgit.freedesktop.org/xorg/app/xrandr/tree/xrandr.c#n3101

       /*
        --------------------> Time -------------->

                          +--------------------+
         Video            |  Blanking          |  Video
                          |                    |
        ----(a)---------->|<-------(b)------->|
                          |                    |
                          |       +--------+   |
                          |       | Sync   |   |
                          |       |        |   |
                          |<-(c)->|<-(d)->|   |
                          |       |        |   |
        ----(1)---------->|       |        |   |
        ----(2)------------------>|        |   |
        ----(3)--------------------------->|   |
        ----(4)------------------------------->|
                         |       |       |   |

        -----------------\                   /--------
                         |                   |
                         \-------\       /---/
                                 |       |
                                 \-------/
     (a) - h_active
     (b) - h_blanking
     (c) - h_sync_offset
     (d) - h_sync_width
     (1) - HDisp / width
     (2) - HSyncStart
     (3) - HSyncEnd
     (4) - HTotal
       */

       assert(hTotal > hSyncEnd);
       assert(hSyncEnd > hSyncStart);
       assert(hSyncStart > width);
       assert(vTotal > vSyncEnd);
       assert(vSyncEnd > vSyncStart);
       assert(vSyncStart > height);

       mode->pixel_clock = dotClock / 1e3;
       // 640x480 @ 75Hz (VESA) hsync: 37.5kHz
       // Modeline "String des" Dot-Clock HDisp HSyncStart HSyncEnd HTotal VDisp VSyncStart VSyncEnd VTotal
[options]
       // ModeLine "640x480"    31.5   640  656  720  840     480  481  484  500
       //                                 16  64  <200         1    3   <20
       mode->h_active = width;
       mode->h_blanking = hTotal - width;
       mode->h_sync_offset = hSyncStart - hTotal;
       mode->h_sync_width = hSyncEnd - hSyncStart;

       mode->v_active = height;
       mode->v_blanking = vTotal - height;
       mode->v_sync_offset = vSyncStart - height;
       mode->v_sync_width = vSyncEnd - vSyncStart;
```

**Refactoring Doc - [https://docs.google.com/document/d/1L8lz7u2uj6MrzSQv4b1Vk6Rmic26okyRklOju5IWLYA/edit](https://docs.google.com/document/d/1L8lz7u2uj6MrzSQv4b1Vk6Rmic26okyRklOju5IWLYA/edit)**

**Mapping to Spartan 6**

**Input**

**Generating the bit clock**

**IDDR**

**ISERDES**

**IDELAY + phase detector**

**BitSlip**

**Symbol decoding**

- **proper decoding**
- **lookup table**
  - **10 bit LUT == 2 * 5 bit LUT per out bit**

**Output**

**OSERDES**

**Hamsters crazy 1080p - [http://hamsterworks.co.nz/mediawiki/index.php/Spartan_6_1080p](http://hamsterworks.co.nz/mediawiki/index.php/Spartan_6_1080p)**