Video Display Unit (VDU)

Historically derived from Cathode Ray Tube (CRT) technology

A CRT scanned the display incessantly, so needed a real time stream of pixel data. Although in principle LCDs could be 'abused' here, the standard interface has been retained.

Analogue TV & Monitors

For many decades televisions were analogue devices which worked as follows:

- A 'spot' was raster-scanned across the screen, 'slowly' in one direction (traditionally left-to-right when viewed from in front) then rapidly in the other direction to its starting point ('flyback').
- Each scan line was displaced by a 'fixed' distance, stepping between sweeps (traditionally from top to bottom); after sufficient scan-lines had been drawn the spot was rapidly returned to the top line.
- During these sweeps the spot's intensity was varied to 'shade' different intensities. Before, during and after flyback – both horizontal and vertical – the spot was 'blanked' (i.e. set to minimum intensity – black) so the flyback was invisible.
  - Blanking covered an interval from near the end of a line until the start of the next, and for several lines’ time for vertical flyback.
  - Colour could be provided by directing three, independently controlled spots onto adjacent (red, green, blue) phosphors.

The display generated is digital in the vertical dimension (it has discrete scan lines) but analogue horizontally as the spot sweeps continuously and its intensity is varied in real time.

For a television it is important that the picture is displayed at the same rate as it is broadcast! To achieve this, synchronisation signals are sent to initiate each flyback. The display has Phase-Locked Loops (PLLs) which regulate its scan rates and these are governed by these 'sync. pulses'. Blanking is initiated some time before the sync. pulse (known as the ‘front porch’) and persists for some time afterwards (the ‘back porch’).

Horizontal and vertical synchronisation is the same, in principle, although the vertical timing is much slower. The interval between vertical syncs. sets the frame rate of the display.

Analogue broadcast TV is an old standard and has/had relatively long blanking intervals when compared with the active video time.

Computer Monitors

CRT computer monitors were derived from TV technology. The more modern devices have proportionately smaller blanking times so most of the time the interface is carrying active (useful) signals, but the principle is the same.

An aging, though still useful, interface is the VGA which has analogue signals driving the colour intensities and digital (true/false) sync. indicators.

The use of analogue signals necessitates Digital to Analogue Converters (DACs) for the colour signals and makes electrical noise pick-up more of a problem.

VDU controller

The VDU controller’s job is to:

- Generate the timing signals for active, blanking and sync. phases
- Generate addresses and read the frame store memory to determine pixel values (colours) when appropriate
- Serialise the pixel values and send them to the display at the pixel rate

It is a fairly simple state machine although the various parameters may be programmable for different display hardware and screen resolutions.
Digital Visual Interface (DVI)

- Primarily digitally coded channels
- Differential signal pairs in each colour
- Two channels (available) to increase potential bandwidth
- Digital communications for monitor type/status/control
- Analogue signals for backwards compatibility

High-Definition Multimedia Interface (HDMI)

There are various DVI standards: DVI-A is a backward-compatible Analogue interface, DVI-D is the Digital form and DVI-I Integrates the two. In some digital interfaces, having two parallel links improves bandwidth allowing higher definition displays.

DVI-D

High-speed digital data is conveyed on multiple serial channels. Each channel comprises a Current Mode Logic (CML) twisted-wire differential pair; this helps to improve noise immunity. The data is encoded using a form of 8b/10b encoding.

Data is sent uncompressed and in real time, thus the general pattern of the output scan is similar to that used for earlier displays such as CRTs.

Display Data Channel (DDC)

With the economic availability of LCD (Liquid Crystal Display) flat-panel displays came a wider range of displays. In particular, the typical aspect ratio of displays has moved from 4:3 to 16:9. Rather than distorting the picture, a better solution is to output the display in an appropriate form but this requires the computer to be aware of the type of display.

The first forms of data sensing merely detected the monitor type. Current communications are more sophisticated with the computer communicating with an embedded controller on the monitor allowing the downloading of information on a monitor’s aspect ratio, resolution, orientation et cetera. If you are sufficiently interested you can look up “EDID” (Extended Display Identification Data). With some it is also possible to write commands to the monitor, for instance to control brightness or contrast.

The usual DDC is based on the two-wire I²C bus which allows fairly low bandwidth communication using a bidirectional serial protocol. This only requires a small addition to the connector and wiring requirement.

The information available from a monitor can identify the manufacturer and model as well as the different resolutions which are supported, the timing characteristics, colour resolution etc. There may be a ‘preferred’ mode which the monitor is designed to use.

HDMI (High-Definition Multimedia Interface)

HDMI is basically similar to DVI-D, only providing a single set of digital channels (no analogue). HDMI uses the blanking period between active video scans to encode control and other information, such as audio channels.

The data may be encoded in ways other than RGB (e.g. YCrCb).

A single, serial Consumer Electronics Control channel is also included to carry data such as that from ‘remote control’ handsets.

8b/10b encoding

In summary, when sending data across a synchronous serial line it is necessary to include enough information for the receiver to recover the clock (to discriminate between adjacent bits) as well as read the data itself. Clearly a pure binary signal is not adequate as it may consist of many consecutive ‘0’s or consecutive ‘1’s.

Many coding schemes have been devised. The link can only be switched at a certain maximum rate; to get the best useful bandwidth, a scheme needs to provide ‘enough’ information to recover the clock but not so much redundancy that the data rate is compromised.

8b/10b is one such scheme which codes 8 bits of data into 10 bit-time symbols (i.e. has a constant 25% overhead). It was first patented (now expired) by IBM in the 1980s.

An important property is that it has DC balance – meaning that, averaged over time, the symbols contain the same number of ‘0’s and ‘1’s. This requires two possible codes per 8-bit byte.

When symbols with insufficient transitions for clock recovery are discarded there are 268 legal codes, which allows any 8-bit data value to be sent plus allowing some control codes (“K-codes”) for the link (which the user need not know about).

8b/10b is in common use, including for DVI, PCI Express, Infiniband, SATA …
Frame store

Based on a 2D array of memory (frame store) with a ‘numeric’ representation of a pixel’s colour.

Frame store

The display is made of pixels (‘picture elements’) which are ‘dots’; typically these are rectangular and preferably more-or-less square. The screen comprises a 2D array of pixels at a particular resolution (vertical & horizontal).

An LCD has physical pixels which determine its maximum resolution. Lower resolution is possible by shading adjacent groups of pixels in the same way: for example a square of four physical pixels could represent a single logical one. If the mapping is non-integer then some distortion may occur.

A standard, but now ‘low’, resolution display is the 640 × 480 VGA (Video Graphics Array). This is specified for the older 4:3 monitor aspect ratio.

The pixel shade/colour is held in a memory called a frame store. Pixels are read successively from the frame store and serialised onto the display. A complete frame refresh is done frequently enough to allow successive frames to give the impression of movement and to avoid disturbing flickering. For computer monitors typical frame rates are in the region 50-100 Hz.

Colour displays are now standard. Each pixel has a colour which is specified by a number of bits. The usual representation for computers is to code intensities of the colours Red, Green and Blue (RGB) separately. This works because human eyes have a limited range of colour sensors; the only colours we actually perceive are centred in these spectral bands and other colours (such as yellow) are perceived from appropriate mixtures of stimuli (red & green for yellow).

The colour outputs are ‘analogue’ (i.e. multi-levelled) outputs where the number of bits used determines how many shades are available. Human eyes are not very sensitive to colour intensities so 8 bits per colour is more than adequate (especially for blue, where perception is worse). Eight bits is, of course, a convenient number for digital computer implementation.

Having three colours is less convenient, so often the entire pixel is mapped into 32 bits; the extra 8 bits can find other uses which need not concern us.

Frame store

Addressing

Note: the traditional address mapping is to have the lowest address at the top-left corner and increment addresses in rows. Thus the x axis runs left to right and the y axis top to bottom.

To move down one pixel (i.e. y := y + 1) requires adding the length of a row to the address.

Also note that if (for example) we have (say) an ARM system with 32 bits/pixel each pixel would occupy four addresses, so moving right one place (x := x + 1) would require adding 4 to the address. If the frame store width was 1024 pixels, moving down one pixel means adding 4 × 1024 = 4094 to the address.

To calculate the address of pixel (x, y):

\[
\text{address} = \text{screen_start_address} + (y \times \text{width_in_pixels}) + x \times \text{bytes_per_pixel}
\]

Typical screen widths (e.g. 640, 1024, 1280) are intended to make these multiplications easy.

A frame store can be larger than the displayed area, although this may waste some memory. It could be made smaller, too, but that would make little sense!
Frame store accessing & bandwidth

Frame store can occupy significant memory. Remember *doubling* the linear resolution multiplies the number of pixels by four.

 Pixels need to be read many times per second to keep the display stable. This impacts:
- The output rate to the DVI (or whatever).
- The need for RAM access to the frame store.

Frame store bandwidth is critical.

### Screen update

Although it is not germane to the drawing process, the frame store is also constantly being read by hardware which is updating the display. This shares access to (typically Time Division Multiplexing) the frame store memory. Memory accesses are relatively slow so frame store bandwidth is always an ‘issue’, made worse as the screen resolution increases.

As it has to be shared, the frame store may not be available exactly when you want it. This influences the interface design. The highest priority for access goes to the VDU read-out because if that fails to meet its real-time constraint there will be glitches visible on the screen. More than one other device may share access too: for example in the lab, both the microprocessor and the graphics accelerator compete for the remaining bandwidth.

### Double buffering

In a system which may animate a display there is a conflict between using the frame store for what can currently be seen and the future picture under construction. This is typically resolved by double buffering: having a larger-than-needed frame store and displaying from one area whilst drawing in another.

In the absence of dual-port memory the accesses either must interleave in time (a typical solution) or two (smaller) separate and switchable frame store memories are needed (expensive).

### Bandwidth requirements

#### Some sums ... and sensible approximations

Let’s take a ‘High Definition’ (HD) display resolution of 1920×1080 pixels with 4 bytes per pixel. This requires 1920×1080×4 = 8294400 bytes of storage.

Rather than reach for a calculator, let’s rough it out.

‘Almost 2000’ × ‘just over 1000’ is going to be around two million pixels so we need 2M×4 = 8 MB of frame store.

Let’s say¹ this supports a frame rate of 50 Hz: it has to be copied to the display 50 times a second, so there is a bandwidth requirement of around 400 MB.s⁻¹.

Note: that’s megabytes, not megabits. Minimum. It doesn’t allow for other data, pauses for blanking, sync. etc.

If you want a bit rate, multiply by 8 and add a bit more for overheads: calling it 4 Gb.s⁻¹ won’t be far wrong.

Looked at another way, the pixel rate will be two million times 50 plus whatever the overhead is, so something over 100 MHz – not too scary a frequency on-chip (these days) but quite aggressive on a PCB!

The frame store needs to be read to supply this demand. If a single pixel (32-bit word) were read at this rate the memory would need to cycle in <10 ns; not really feasible for the ‘big’ (multi-megabyte even assuming a single frame store and there could be more than one) memory devices needed. Thus there needs to be a means of increasing the memory bandwidth. Fortunately the read-out patterns are entirely predictable; it’s easy enough to read the frame store at many words wide and then serialise this data.

Also note, if implementing animation, at least, there is another bandwidth requirement to allow concurrent writing of the pixels – and a real-time limit too.

---

1. To keep the numbers easy.
Drawing Straight Lines

An example of mapping an algorithm to hardware.

\[ y = mx + c \]

Line is *aliased* onto pixel array.

---

**How not to plot a line**

Don’t calculate every point independently.

\[ m = \frac{Y_1 - Y_0}{X_1 - X_0} \]

- \( \int(x+0.5) \) rounds \( x \) to the nearest integer

**Problems:**
- Division needed once
- Multiplication needed constantly
- Rounding errors

---

**Anti-aliasing**

Figures drawn in square pixels – especially at low resolution – end up ‘pixelated’; lines look stepped.

Anti-aliasing is a method of blurring these steps. All pixels the theoretical line crosses are shaded but the degree of shading is proportional to how much of the pixel the true line passes through. The line’s colour is blended with the background.

Anti-aliasing requires considerably more calculation and more memory operations (including reading the pre-existing background).
Bresenham’s line algorithm

- Calculate each point iteratively from its predecessor
- Avoid multiplication/division (by using similar triangles)
  - Uses only integers: no rounding problems

Principle

\[
\begin{align*}
  x &= X_0; \\
  y &= Y_0; \\
  \text{plot} \ (x, y); \\
  \text{length} &= X_1 - X_0; \\
  m &= (Y_1 - Y_0) / (X_1 - X_0); \\
  e &= 0; \\
  \text{for} \ (\text{length}) \\
  x &= x + 1; \\
  e &= e + m; \\
  \text{if} \ (e \geq 0.5) \\
  y &= y + 1; \quad // \ y \text{ integer step} \\
  e &= e - 1; \quad // \ \text{Keep} \ |e| < 0.5 \\
  \text{plot} \ (x, y);
\end{align*}
\]

Integer code

\[
\begin{align*}
  x &= X_0; \\
  y &= Y_0; \\
  \text{plot} \ (x, y); \\
  dx &= X_1 - X_0; \\
  dy &= Y_1 - Y_0; \\
  e &= -dx; \quad // \ \text{Starting offset} \\
  \text{for} \ (dx) \\
  x &= x + 1; \\
  e &= e + 2*dy; \\
  \text{if} \ (e \geq 0) \quad // \ \text{Easy compare} \\
  y &= y + 1; \\
  e &= e - 2*dx; \\
  \text{plot} \ (x, y);
\end{align*}
\]

Octants

The foregoing assumes that the line is in the shaded octant, shown here. If it is not, the same approach can be followed with some slight variations.

In this example, x is incremented and y is incremented conditionally. For the octant immediately below the x axis, x is incremented and y is conditionally decremented. As long as the coordinates are modified in the correct way it the signs of the internal variables are irrelevant.

Similarly, if the slope of the line is >1 (i.e. 'steeper than 45°') then x and y are exchanged. A similar transformation can be applied if the line is going 'right' or 'down'.

Similar triangles

The gradient ('m') of a step from one pixel to the next is derived from the vertical/horizontal distances between end points. Although 'm' is typically fractional (0 ≤ m ≤ 1) the distances between endpoints are integers.

Thus, when considering whether the y coordinate should change, instead of thinking of little steps (1, m) we can think of big ones (2dx, 2dy) and the decision will still be the same.

(The extra factor of 2 is convenient because we want to step when half-way to round to the nearest pixel and this avoids the \(\frac{1}{2}\)).

Optimisation

There is another optimisation which reduces the length of the loop by simplifying the 'plot' operation. Instead of translating coordinates on each iteration, simply work out the address of the starting point and retain that. Using the assumptions of 'one address per pixel' and '640 pixels per line', the following translations take place:

\[
\begin{align*}
  x &= x + 1 \Rightarrow \text{address} = \text{address} + 1 \\
  y &= y + 1 \Rightarrow \text{address} = \text{address} + 640
\end{align*}
\]

The plot no longer needs to do any translation, just the store.

A disadvantage of this method is that running off the edge of the frame store is not apparent, as it may be if clipping the x and y coordinates.

If you have more than one pixel/word in the frame store (as in the lab.) then one can speed up drawing by writing several pixels at once. These pixels must be in the same word and so will form a horizontal group. This is not very useful when drawing single lines because there will often not be several adjacent pixels within the same word.

It is very useful when filling areas (e.g. clear screen) and similar (e.g. character drawing) where it can reduce drawing times by (e.g. 4x).
Identifying parallelism is a good plan: e.g. Bresenham’s line algorithm

2 clocks/iteration

```vhdl
x <= X0;
y <= Y0;
dx <= X1 - X0;
dy <= Y1 - Y0;
e <= -dx;
for (dx)
    plot(x,y);
x <= x + 1;
e <= e + 2*dy;
if (e >= 0)
    y <= y + 1;
e <= e - 2*dx;
plot(x,y);
```

1 clock/iteration

```vhdl
x <= X0;
y <= Y0;
dx <= X1 - X0;
dy <= Y1 - Y0;
e <= -dx;
for (dx)
    plot(x,y);
x <= x + 1;
if (e + 2*dy >= 0)
    e <= e + 2*(dy - dx);
else
    e <= e + 2*dy;
plot(x,y);
```

Also note the **pipelining** here: plot overlaps with the next pixel calculation.

In the second example the critical path is likely to be longer (‘if’ calculation followed by multiplexer) but not much worse (multiplexers are quick).

---

**Parallelism**

Probably the biggest ‘mistake’ made by people starting to develop HDL code is to think *serially*, as it a conventional (imperative) programming language. In C, Java, assembly language etc. statements can be viewed as executing one after the other … because they need to (at least in principle).

In hardware the only needs are due to dependencies and resources – and resources shouldn’t be too much of an issue within this lab. Thus statements need to be mapped into time slots but as many statements as possible can go in the same time. This leads to a much faster implementation than a simple one-statement-per-clock machine.

The number of serial processing steps which take place in a single cycle (i.e. the **critical path** length) also concerns the designer; however the cycle is generous in the lab. so it is not likely to be a major concern when describing logic.

When developing your own code, **design** it before you implement. Plan what should happen (e.g. on a piece of paper) in each clock cycle.

Pay attention to which values are latched. A common problem is that a value is only available after a clock edge when you want it in the current cycle. The choice is then whether to derive the signal combinatorially so that it is available a bit earlier or whether to start work a cycle earlier. See the problem on the right.

---

**Problem**

Fill in the timing diagram for this module.

```vhdl
reg [3:0] counter;
reg carry;
always @ (posedge clk)
    if (en && carry_in) // Hint on fn. of ‘carry’
    begin
        if (counter == 9)
            begin
                counter <= 0;
carry <= 1;
            end
        else
            begin
                counter <= counter + 1;
carry <= 1;
            end
    end
```

The circuit is unlikely to be useful. Rewrite the Verilog in **at least one** way to do what the designer presumably intended.