Texture Mapping Unit, Revision 2

Sébastien Bourdeauducq

April 2010

1 Presentation

The texture mapping unit supports the following operations:

- 2D texture mapping on a tesselation of equally-sized rectangles.
- Operates only on rectangular rendering primitives (no triangles).
- Fixed-point texture coordinates (1/64 pixel resolution).
- Configurable bilinear texture filtering.
- Configurable texture wrapping (only dimensions that are a power of 2 are supported for wrapping).
- Configurable texture clamping (any dimension).
- Up to 2047x2047 texture size.
- Up to 2047x2047 output buffer resolution.
- Chroma key filtering.
- Fading to black.

The core processes 16-bit RGB565 progressive-scan framebuffers, accessed via FML links with a width of 64 bits and a burst length of 4.

The vertex data is fetched using a 32-bit WISHBONE master. Connecting this bus to the WISHBONE-to-FML caching bridge allows the mesh data to be stored in cost-effective DRAM.

For controlling the core, a CSR bus slave is also implemented.

2 Configuration and Status Registers

Registers can be read at any time, and written when the core is not busy. Write operations when the busy bit is set in register 0, including those to the control register, are illegal and can cause unpredictable behaviour.

Addresses are in bytes to match the addresses seen by the CPU when the CSR bus is bridged to Wishbone.

2.1 Parameters and control

Offset	**Access**	**Default**	**Description**
0x00	RW	0	Control register. Bit 0: Start/Busy. Bit 1: Enable chroma key.
0x04	RW	32	Number of mesh rectangles in the X direction (which is the number of mesh points minus one, and also the index of the last mesh point).
0x08	RW	24	Number of mesh rectangles in the Y direction.
0x0C	RW	63	Brightness, between 0 and 63. The components of each pixel are multiplied by $\frac{(n+1)}{64}$ and rounded to the lowest integer. That means that a value of 0 in this register makes the destination picture completely black (because of the limited resolution of RGB565).
0x10	RW	0	RGB565 color used as chroma key. Texture pixels with this color will not be drawn if the "chroma key" flag in the control register is set.
0x14	RW	0	Vertex mesh address. The address must be aligned to a 64-bit boundary.
0x18	RW	0	Texture buffer address. The address must be aligned to a 16-bit boundary.
0x1C	RW	512	Texture horizontal resolution.
0x20	RW	512	Texture vertical resolution.
0x24	RW	0x3ffff	Binary mask ANDed to the fixed-point X texture coordinate during interpolation. This mask can be used to control texture wrapping and filtering.
0x28	RW	0x3ffff	Binary mask ANDed to the fixed-point Y texture coordinate during interpolation.
0x2C	RW	0	Destination framebuffer address. The address be aligned to a 16-bit boundary.
0x30	RW	640	Destination horizontal resolution.
0x34	RW	480	Destination vertical resolution.
0x38	RW	0	X offset added to each destination pixel, allowing the use of the TMU as a blitter. Negative offsets in two's complement format are supported.
0x3C	RW	0	Y offset added to each destination pixel.
0x40	RW	16	Width of each destination rectangle.
0x44	RW	16	Height of each destination rectangle.
0x48	RW	63	Opacity (alpha) used when drawing in the destination framebuffer. 0 = high transparency, 63 = totally opaque. A value of 63 saves memory bandwidth by removing the need to read-modify-write the destination framebuffer.

3 Interrupts

The TMU is equipped with one active-high edge-sensitive interrupt line.

An interrupt is triggered when a texture mapping is done and all resulting data has been sent through the bus master.

4 Encoding the vertex data

The core supports a maximum mesh of 128x128 points. The address of the point at indices (x, y) in the mesh is, regardless of the actual the number of mesh points:

$$base + 8 \cdot (128 \cdot y + x)$$

This means that the mesh always has the same size in memory.

Each point is made up of 64 bits, with the 32 upper bits being the X coordinate and the 32 lower bits the Y coordinate, in fixed-point two's complement signed format with 6 bits of fractional part.

Exactly 128kB are used by the mesh.

5 Architecture

The texture mapping unit has a deeply pipelined architecture following the "dataflow" model. The first stage fetches vertex data, which are in turn passed to the second stage which computes operands for the division, done in the third stage, used in linear interpolations, etc.

5.1 Handshake protocol between pipeline stages

Because pipeline stages are not always ready to accept and/or to produce data (because, for example, of memory latencies), a flow control protocol must be implemented.

The situation is the same between all stages: an upstream stage is registering data into a downstream stage. During some cycles, the upstream stage cannot produce valid data and/or the downstream stage is processing the previous data and has no memory left to store the incoming data.

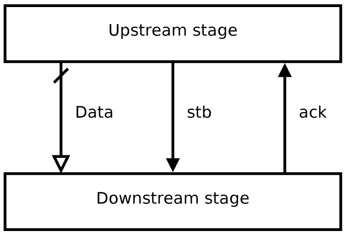

Figure 1: Communication between two pipeline stages.

Appropriate handling of these cases is done using standardized `stb` and `ack` signals. The meaning of these is summarized in this table:

stb	ack	Situation
0	0	The upstream stage does not have data to send, and the downstream stage is not ready to accept data.
0	1	The downstream stage is ready to accept data, but the upstream stage has currently no data to send. The downstream stage is not required to keep its ack signal asserted.
1	0	The upstream stage is trying to send data to the downstream stage, which is currently not ready to accept it. The transaction is *stalled*. The upstream stage must keep stb asserted and continue to present valid data until the transaction is completed.
1	1	The upstream stage is sending data to the downstream stage which is ready to accept it. The transaction is *completed*. The downstream stage must register the incoming data, as the upstream stage is not required to hold it valid at the next cycle.

It is not allowed to generate `ack` combinatorially from `stb`. The `ack` signal must always represent the current state of the downstream stage, ie. whether or not it will accept whatever data we present to it.

6 Bilinear filtering

6.1 Principle

Bilinear filtering works by adding 6 extra bits of precision to the texture coordinates, which become fixed-point non-integer coordinates.

Those bits are used to compute a weighted average of 4 neighbouring texture pixels when the interpolated texture coordinates are not integer.

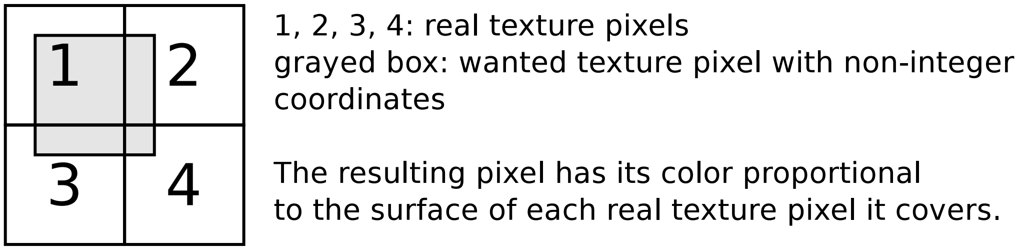
1
2
3
4
1, 2, 3, 4: real texture pixels
grayed box: wanted texture pixel with non-integer coordinates
The resulting pixel has its color proportional
to the surface of each real texture pixel it covers.

Figure 2: Principle of bilinear texture filtering.

6.2 Pixel distribution in the cache

Because of performance requirements, all four texture pixels must be fetched in one clock cycle. It is therefore relevant to examine their distribution in the cache, to determine what cache architecture should be used.

6.2.1 General case, middle of texture

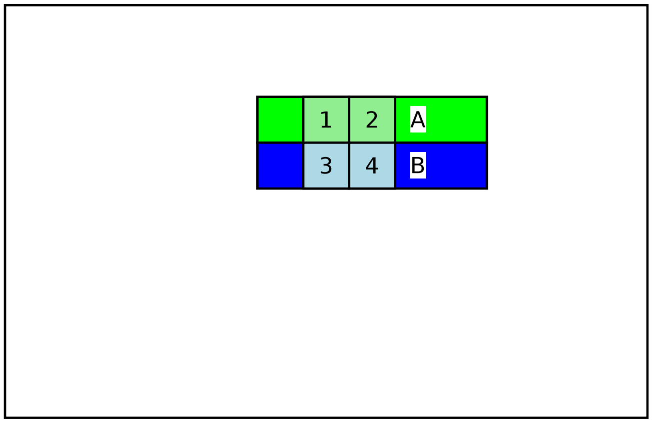

Figure 3: Most common case, pixels 1 and 2 are in the same cache line.

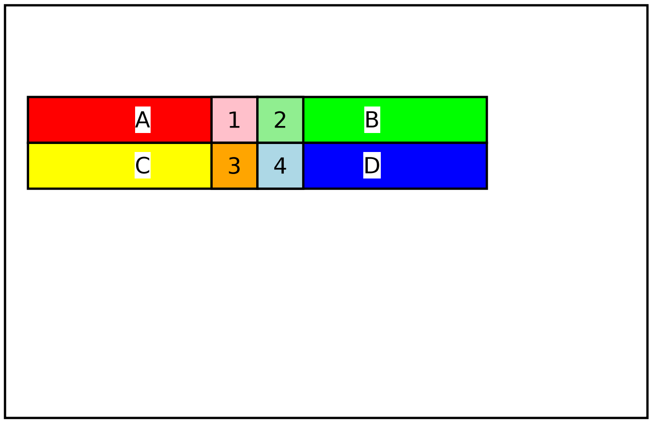

Figure 4: Pixels 1 and 2 are in different cache lines.

6.2.2 With clamping

When pixels go out of the texture, clamping can induce cases where two or four pixels merge.

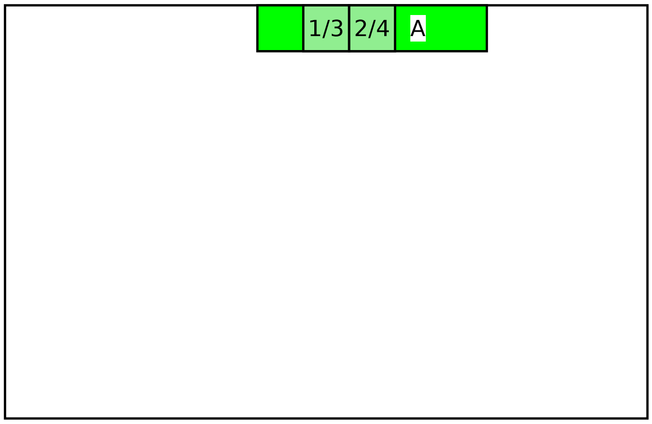

Figure 5: Example of pixels merging because Y1=-1.

6.2.3 With wrapping

Wrapping can cause pixels split across the texture. This can happen horizontally, vertically or both.

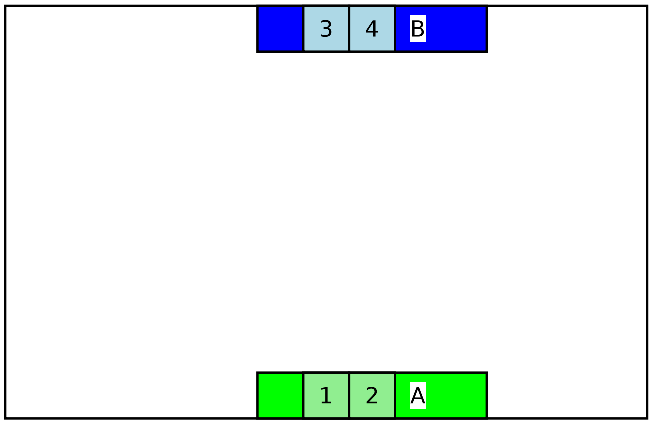

Figure 6: Vertical wrapping.

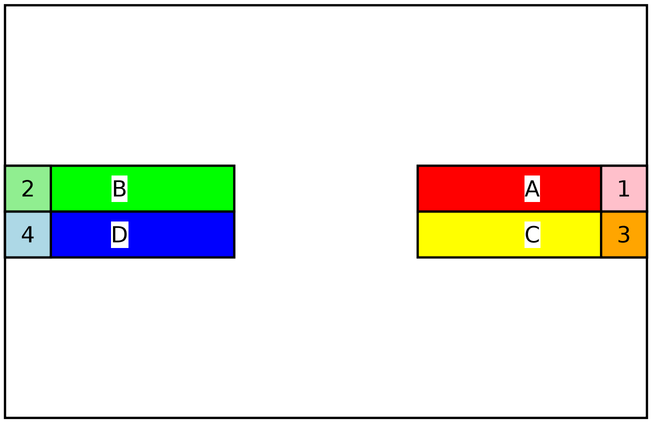

Figure 7: Horizontal wrapping.

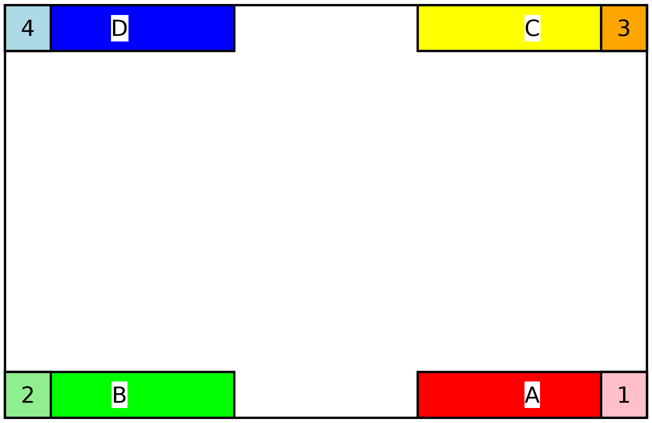

Figure 8: Wrapping in both directions.

6.3 Cache architecture

6.3.1 Design options

We have two options for designing the cache:

- four separate caches (one for pixel 1, one for pixel 2, etc.)
- a shared cache capable of looking up 4 pixels at a time

Misses happening in the ways for pixels 1, 2, 3 and 4 are likely to be correlated, so a shared cache architecture is chosen to minimize memory bandwidth.

6.3.2 Avoiding cache conflicts

There could be replacement conflicts that have, at least, a detrimental impact of performance.

Such conflicts happen when:

- (Figure 3) lines A and B collide. This happens when:

$$\text{hres} \equiv 0 \pmod{\text{csize}}$$

- (Figure 4) lines A and B collide or lines C and D collide. This does not happen unless the cache only contains one line, which is not a practical case.
- (Figure 4) lines A and C or lines B and D collide. This is the same case as for Figure 3.
- (Figure 6) lines A and B collide. This happens when:

$$\text{hres} \cdot (\text{vres} - 1) \equiv 0 \pmod{\text{csize}}$$

- (Figure 7) lines A and B or lines B and C collide. This happens when:

$$\text{hres} - 1 \equiv 0 \pmod{\text{csize}}$$

In the equations above:

- hres is the horizontal resolution in pixels
- vres is the vertical resolution in pixels
- csize is the total number of pixels the cache can hold. It is equal to the cache size in bytes (not counting the tag memory) divided by 2.

6.3.3 Chosen architecture

The block diagram of the cache architecture is given in Figure 9.

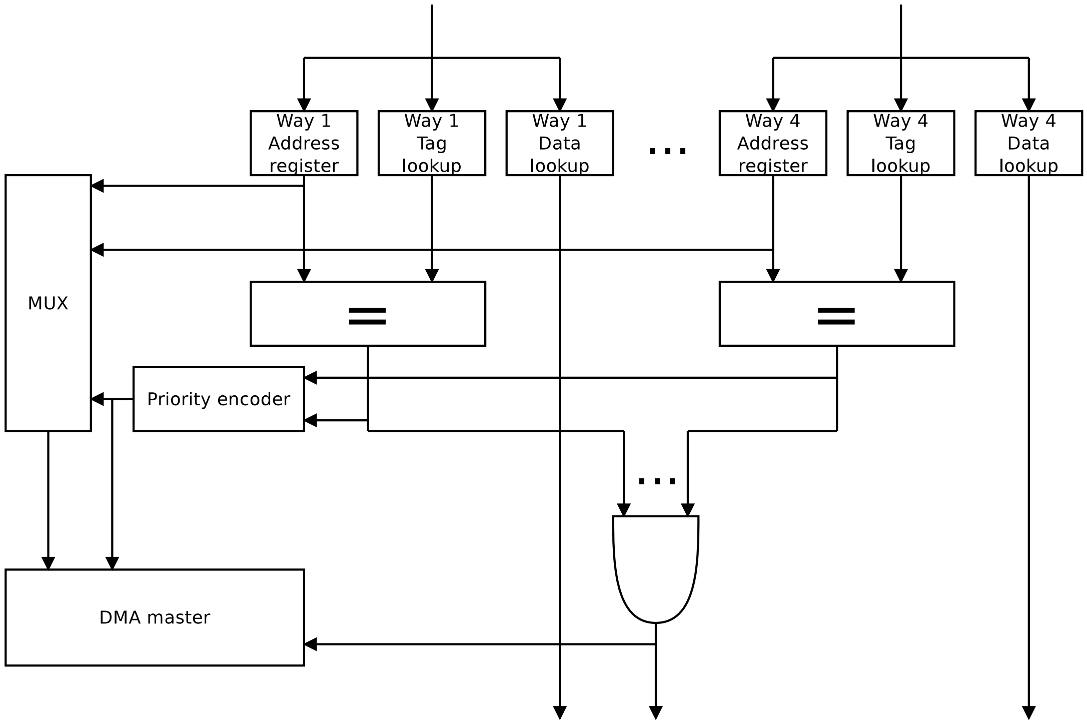

Figure 9: TMU cache architecture for bilinear filtering.

Because cache conflicts can be easily be avoided by choosing the texture resolution carefully, they are not handled by this architecture. **Having a cache conflict results in a lockup of the cache controller**, that requires a reset of the core.

The four-way cache can be efficently implemented on FPGA targets by using two dual-port block RAMs.

6.3.4 Summary

In order to avoid cache conflicts which lock up the core, one must make sure that:

$$\text{hres} \not\equiv 0 \pmod{\text{csize}}$$

Additionally, if texture wrapping is enabled, one must also make sure that:

$$\text{hres} \cdot (\text{vres} - 1) \not\equiv 0 \pmod{\text{csize}}$$

$$\text{hres} - 1 \not\equiv 0 \pmod{\text{csize}}$$

Copyright notice