KUNGL
TEKNISKA
HÖGSKOLAN

TRITA-ICT-EX-2010:90

A performance-driven SoC architecture for video synthesis

Sébastien Bourdeauducq

Stockholm 2010

Master of Science Thesis in System-on-Chip Design
Royal Institute of Technology
Department of Software and Computer Systems

TRITA-ICT-EX-2010:90

KTH, Stockholm 2010

Abstract

Commercial system-on-chips with advanced graphics acceleration capabilities are becoming ubiquitous today. However, in contradiction with the open source idea, little is known about the details of their architecture and implementation, as they are usually covered by trade secrets.

Fostered by the falling costs of high-density FPGAs, our thesis project encompasses researching, developing and implementing the key points of the architecture of an open source and comprehensive system-on-chip with competitive yet reasonable graphics capabilities. The chosen target application is the synthesis of visual effects similar to those produced by the popular MilkDrop visualization plug-in for Winamp.

Our system-on-chip design consists principally of a custom bus infrastructure, a custom DDR SDRAM memory controller, a microprocessor core, and custom graphics accelerators for texture mapping and floating point processing.

Our base microprocessor system is capable of running Linux (without MMU) and outperforms a Microblaze-based solution tested in similar conditions by a 15 to 35% increase in speed of execution. For our video synthesis application, our texture mapping accelerator achieves an average fill rate of 44 megapixels per second and our floating point processing unit provides in excess of 70 million floating point operations per second. Everything, including I/O peripherals (AC97 audio, Ethernet, RS232 UART, GPIO), is implemented on a Virtex-4 XC4VLX25 FPGA, where it utilizes about 80% of the resources.

Finally, we have successfully developed an embedded video synthesis program that leverages the possibilities of our hardware architecture to permit the live rendering of many MilkDrop effects in 640x480 resolution at 30 frames per second.

TRITA-ICT-EX-2010:90

Acknowledgments

First, I would like to express my gratitude to Professor Mats Brorsson, my supervisor and examiner at the Royal Institute of Technology, for having the open-mindedness of letting me write my thesis on this subject and for his help and advice with it.

I would also like to thank Lattice Semiconductor for opening the source code of their LatticeMico32 processor core.

Special thanks go to all the people who are indirectly involved with this Master's thesis project: Henry de Beauchesne (Xilinx) for getting me started with high-end FPGA tools, Shawn Tan (Aeste Works (M) Sdn Bhd) for his help with understanding the WISHBONE bus, Gregory Taylor (NASA's Jet Propulsion Laboratory) for letting me know that they were using parts of my code in the development of a communications system to be put on board the international space station, Takeshi Matsuya (Keio University) for his work on the port of Linux to the system-on-chip described herein, Michael Walle for developing support of the system-on-chip in the QEMU emulator and Wolfgang Spraul (Sharism at Work Ltd.) for proposing me an agreement for manufacturing devices using the system-on-chip design.

Thanks to the Eidôlon music band (Rheims, France), for whom I wrote my first PC-based video synthesis program in 2005, which has been a source of inspiration for this project.

Finally, I would like to thank all the researchers who have retained their copyright on their papers (or have put them in the public domain) and distribute them online for everybody to download freely (incidentally in accordance with the principle of free exchange of information from the KTH ethics policy). This in spite of the default agreement of many publishers such as the IEEE, which asks authors to assign their copyrights to the publishers so the latter have the exclusive permission to sell the download of documents that they did not write, without giving back to the authors, at a price supposedly meant to cover publishing expenses but which is not justified by today's low costs of network bandwidth and servers.

Thanks to these researchers, I have been able to access quality scientific literature before I went to a university, from which I have learned a lot. Even throughout the writing of this Master's thesis, papers freely available online enabled greater productivity as access to them was much faster.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The open source model supports the idea that any individual, if he or she has the required level of technical knowledge, can realistically use, share and modify the design of a technical system. During the nineties, this development model gained popularity in the software world with, most notably, the Linux operating system. But it was not viable for complex SoCs until a few years ago, because the cost of prototyping semiconductor chips is prohibitive and field programmable gate arrays (FPGAs) used to be too slow, too small, and too expensive. System-on-chip design and hands-on computer architecture therefore remained a field reserved to well-funded academia and research and development laboratories of companies of a significant size and wealth, who had access to large FPGA clusters or even semiconductor foundries.

But the cost of FPGAs is falling (this was already the case between 1985 and 1994 [24] and the trend has continued since then) and relatively fast and high-density devices are today becoming available to the general public. For an example of this falling cost (and increasing densities and speed), we will mention the Ikos Pegasus application specific integrated circuit (ASIC) emulator, whose insides are depicted in figure 1.1. The LatticeMico32 CPU core used in the system-on-chip described in this thesis occupies alone 60% of the resources of one of the XC4036XL FPGAs of this device, and runs at 30MHz. The Ikos Pegasus was a state-of-the-art device a decade ago. It consumes up to 3 kilowatts of power, weights dozens of kilos and probably costed the equivalent of several millions of SEK. The same CPU core now occupies about 15% of a modern FPGA costing less than 500 SEK, where it runs in excess of 100MHz.

This evolution makes it possible to implement complex high-performance system-on-chips (SoC) that can be modified and improved by anyone, thanks to the flexibility of the FPGA platform.

This Master's thesis introduces Milkymist^{TM} [6], a fast and resource-efficient FPGA-based system-on-chip designed for the application of rendering live video effects during performances such as concerts, clubs or contemporary art installations. Such effects are already popularized by artists known as "video jockeys", or "VJs". VJing is commonly done with a PC and computer software such as GrandVJ [5] or

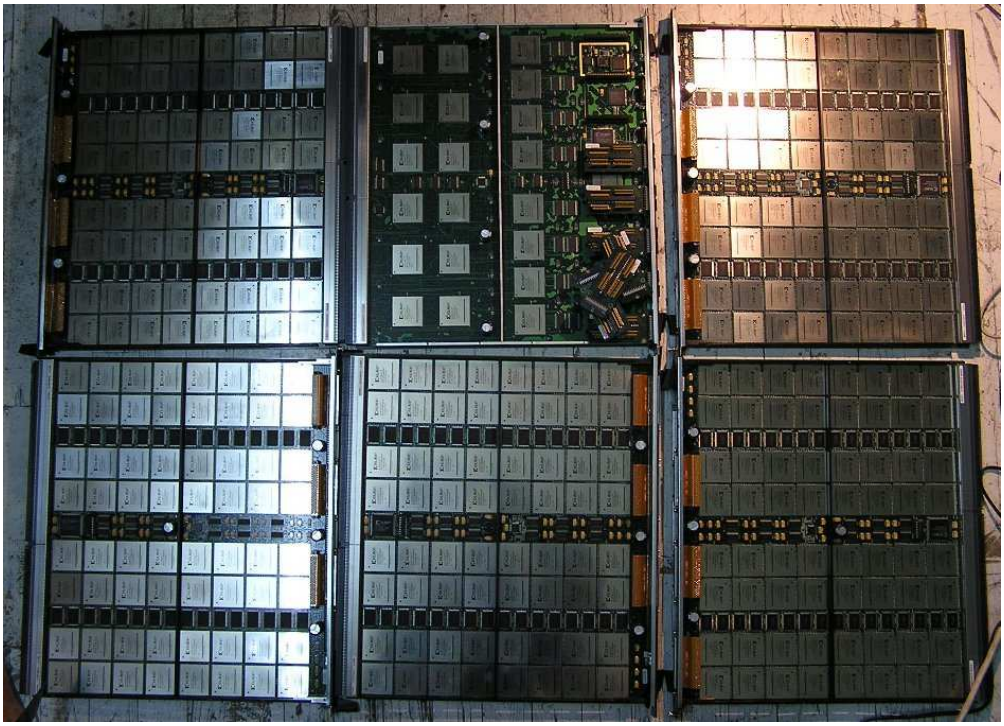

Figure 1.1. FPGA boards of the Ikos Pegasus ASIC emulator (ca. 1999).

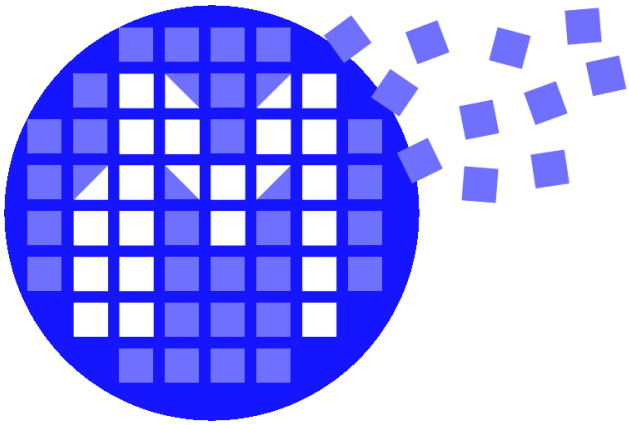

Figure 1.2. Project logo.

Resolume [11]. However, this approach has some drawbacks and using an embedded device instead would be interesting:

- A device of very small size and weight is possible, which is convenient in mobile or temporary setups.
- Boot and set-up time (launching the software) can be greatly reduced (to a few seconds).
- Many interfaces for interactive performances (MIDI, DMX, video input, low-level digital I/O for user sensors) can be integrated. By comparison, the equivalent PC-based solution would be expensive and bulky.

Besides the fact that this is an interesting, creative and popular application, it is also demanding in terms of computational power and memory performance. Such a project would also be a proof that high performance open source system-on-chip design is possible in practice; with a view to help, foster and catalyze similar "open hardware" initiatives. As the Milkymist system-on-chip is entirely made of synthesizable Verilog and, for the most part, released under the GNU General Public License (GPL), its code can be re-used by other open hardware projects.

Meeting the performance constraints while still using cheap and relatively small FPGAs is perhaps the most interesting and challenging technical point of this project, and it could not be done without substantial work in the field of computer architecture. This is what this Master's thesis covers.

Chapter 2

Background

2.1 Video synthesis

2.1.1 Overview

MilkDrop [25] (figure 2.1) is a popular open source video synthesis framework that was originally made to develop visualization plug-ins for the Winamp audio player. People have since then ported MilkDrop to many different platforms [32] and made it react to live events, such as captured audio and video [20] (figure 2.2) or movements of a Wiimote remote control [21].

The idea behind the Milkymist project is to implement an embedded video synthesis platform on a custom open source system-on-chip, that is based on the same rendering principle of MilkDrop but with more control interfaces and features. The device built around the system-on-chip should be stand-alone, which means that a graphical user interface for configuring the visual effects should be implemented (figure 2.3).

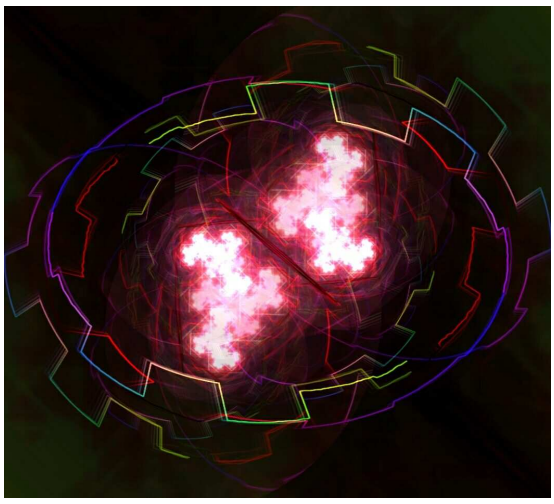

Figure 2.1. Sample video frame from the MilkDrop visual synthesizer.

Figure 2.2. Sample video frame from Visikord, a program mixing live video into MilkDrop.

2.1.2 Principle

General mode of operation

The MilkDrop-like renderer is the most compute and memory intensive process, from which stem most of the technical challenges. We will now get into more details about how the renderer works (figure 2.4).

Rendering is based on a frame buffer on which the steps below are continuously repeated. This repetition is at the origin of many feedback or "fractal" effects.

- The current frame is distorted (zoomed, translated, warped, scaled, rotated...) by texture mapping. This step is described with more detail in section 2.4.
- The frame is darkened (the colors are shifted to black).
- A waveform of the currently played music is drawn. The wave can be drawn linearly (like an oscilloscope), in a circle, etc.
- Borders around the screen are drawn. If the distortion zooms out, the borders will be pulled into the picture (some effects are based on this).
- *Motion vectors* are drawn. Motion vectors are simply a grid of dots, which can be used to generate effects by playing with the distortion.
- The process repeats from the beginning.

These are the basic features of MilkDrop. There are more (custom waves, shapes,...) which are listed on the MilkDrop website [25]. Some other features (such as adding live video) will be added to the Milkymist renderer in the future.

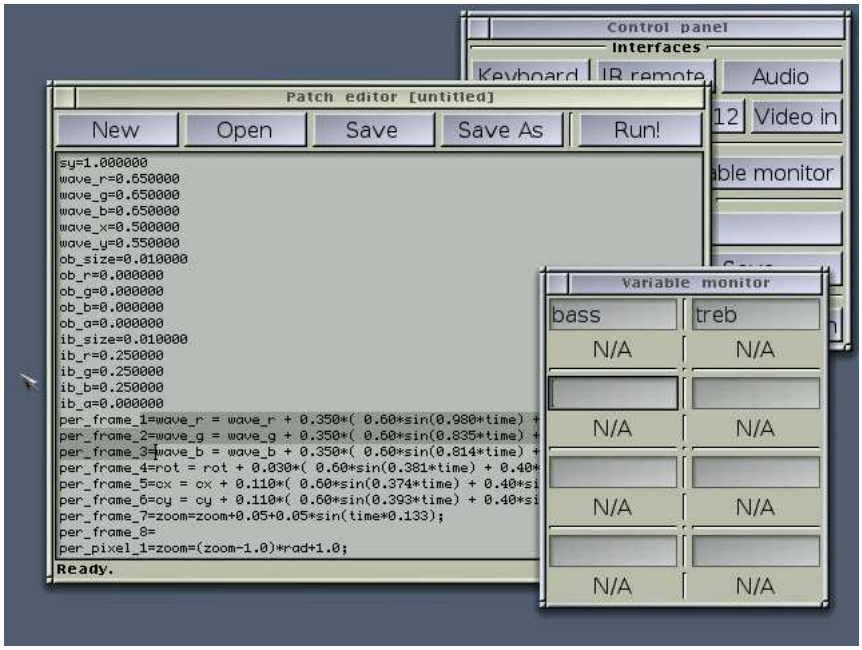

Figure 2.3. The embedded user interface (based on Genode FX [12]) of Flickernoise, the Milkymist VJ application. The patch editor is shown, with per-frame and per-vertex equations.

This process is done on an internal frame buffer whose horizontal and vertical dimensions are a power of 2. This frame buffer is then scaled to the size of the screen in order to be displayed. This brings two features:

- The sizes being a power of 2 allows out-of-bounds texture coordinates to be wrapped (in order to repeat the texture) by simply performing a bitwise AND of the coordinate, instead of the full computation of a division remainder which is a much more expensive operation (even on the traditional GPUs MilkDrop was designed for).

- It enables the implementation of the *video echo* effect: after the internal frame buffer has been drawn to the screen at its nominal dimensions, a zoomed and semi-transparent copy of it can be overprinted.

It must be noted that this two-step process increases the computation time and the consumption of memory bandwidth.

All the steps of the rendering are heavily parameterizable by the user, using a coded format called a *patch* or *preset* which defines the aspect and the interaction forms of a particular visual effect. The listing of a sample patch is given by figure 2.5 and the meaning of the language is explained below.

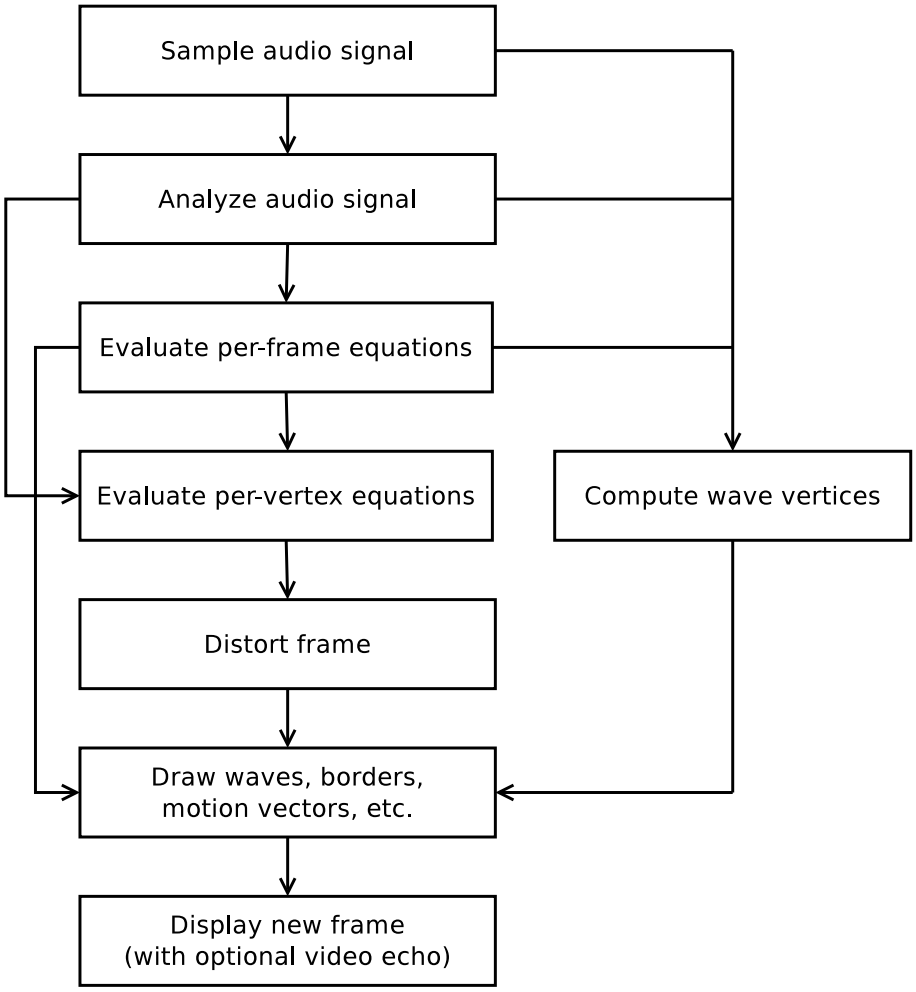

Figure 2.4. Basic MilkDrop rendering flow.

```
fDecay=0.980000
nWaveMode=2
bTexWrap=1
bMotionVectorsOn=0
zoom=1.046000
rot=0.020000
cx=0.500000
cy=0.500000
warp=0.969000
sx=1.000000
sy=1.000000
wave_r=0.600000
wave_g=0.600000
wave_b=0.600000
wave_x=0.500000
wave_y=0.470000
per_frame_1=wave_r = wave_r + 0.400*( 0.60*sin(0.933*time)
  + 0.40*sin(1.045*time) );
per_frame_2=wave_g = wave_g + 0.400*( 0.60*sin(0.900*time)
  + 0.40*sin(0.956*time) );
per_frame_3=wave_b = wave_b + 0.400*( 0.60*sin(0.910*time)
  + 0.40*sin(0.920*time) );
per_frame_4=zoom = zoom + 0.010*( 0.60*sin(0.339*time)
  + 0.40*sin(0.276*time) );
per_frame_5=rot = rot + 0.050*( 0.60*sin(0.381*time)
  + 0.40*sin(0.579*time) );
per_frame_6=cx = cx + 0.030*( 0.60*sin(0.374*time)
  + 0.40*sin(0.294*time) );
per_frame_7=cy = cy + 0.030*( 0.60*sin(0.393*time)
  + 0.40*sin(0.223*time) );
per_vertex_1=sx=sx-0.04*sin((y*2-1)*6+(x*2-1)*7+time*1.59);
per_vertex_2=sy=sy-0.04*sin((x*2-1)*8-(y*2-1)*5+time*1.43);
```

Figure 2.5. Excerpt from the MilkDrop preset "Geiss – Warp of Dali 1" (with some simplifications).

Initial conditions

The patch begins with a series of parameters which are used to initialize the renderer, and many of them are kept constant during the execution of the patch. For example:

- `bMotionVectorsOn=0` turns off the drawing of the motion vectors.
- `nWaveMode=2` selects one of the many ways of drawing the audio waveform.
- `sx=1.000000` and `sy=1.000000` set the X and Y scaling factors of the distortion to 1 (i.e. the frame is initially not scaled).
- `wave_r=0.600000`, `wave_g=0.600000` and `wave_b=0.600000` set the initial RGB color with which the wave is drawn (it is initially grey).

Per-frame equations

Using initial conditions only limits the interaction and evolution possibilities of the patch.

It is therefore possible to make the parameters evolve over time, thanks to the per-frame equations. As their name suggests, the per-frame equations are mathematical expressions that are evaluated at each frame.

The example patch (figure 2.5) shows some of them (the lines beginning with `per_frame`). In this example, they change the wave color over time by modifying the `wave_r`, `wave_g` and `wave_b` values in sinusoidal patterns, as well as the zoom (`zoom`), rotation (`rot`) and center of rotation (`cx` and `cy`).

Per-frame equations can make the patch react to sound, for example through the `bass`, `mid` and `treb` variables that indicate the intensity of the sound in three frequency bands. One of the ideas in Milkymist is to add other variables that can be controlled by the DMX512 and MIDI protocols, enabling the use of a whole range of devices commonly found among musicians (electronic instruments, faders, stage light consoles, joysticks,...) to control the visual effects.

Per-vertex equations

Per-vertex equations are used to fine-tune the distortion applied to the picture.

Indeed, as explained further in section 2.4, the distortion works by using a mesh of control points (vertices) that can be moved to transform the image in many different ways (effects such as zooming, scaling and rotating are implemented by moving the vertices).

Per-vertex equations are thus evaluated at each vertex (whose position can be retrieved through the `x` and `y` variables), and alter the position of that vertex. In the example patch (figure 2.5), the image is locally scaled horizontally and vertically by factors depending on the position of the vertex and on the time, resulting in a twisted visual effect.

As discussed in chapter 5, the floating point computations for each vertex are intensive and required the use of a dedicated co-processor.

2.2 Open source SoC platforms

There is an existing effort to build open source system-on-chips. It is interesting to review these projects in order to look forward to building upon them — possibly adding hardware accelerators or performing other modifications in order to improve performance.

There are many SoC designs available on the Internet, which are more or less mature. The system-on-chip projects listed here meet the following criteria:

- they have been shown to work on at least one FPGA board
- they are released under an open source license
- they comprise a synthesizable RISC CPU
- the CPU is supported by a C and C++ compiler
- they include a RS232 compatible UART (for a debug console)
- they support interfacing to off-chip SDRAM memory

OpenSPARC

OpenSPARC [23] is the well-known SPARC processor of Sun Microsystems which is now released under an open source license and included into a SoC FPGA project.

Implemented on a FPGA, this processor is extremely resource-intensive. A cut-down version of the CPU core only, called the "Simply RISC S1", occupies at least 37000 FPGA look-up tables (LUT) without the caches [28]. This is about twice the logic capacity of the Virtex-4 XC4VLX25 FPGA.

As it turns out, the OpenSPARC architecture is a very complex design which implements a huge number of techniques which increase the software execution speed (instructions per clock cycle). While this is a wise choice for a software-centric processor implemented on a fully custom semiconductor chip, with a FPGA process it is more appealing to keep the software processor simple in order to save resources and make room for custom hardware accelerators, taking advantage of the FPGA flexibility.

GRLIB

GRLIB [13] is a very professional and standard-compliant library of SoC cores. The library features a comprehensive set of cores: AMBA AHB/APB bus control elements, the LEON3 SPARC processor, a 32-bit PC133 SDRAM controller, a 32-bit PCI bridge with DMA, a 10/100/1000 Mb/s Ethernet MAC, 16/32/64-bit DDR SDRAM/DDR2 SDRAM controllers and more.

However, its drawbacks are:

- Code complexity. GRLIB is written in VHDL and makes intensive use of custom types, packages, generate statements, etc.

- Cores are not self-contained. GRLIB defines many "building blocks" that are used everywhere else in the code, making it difficult to re-use code in another project which is not based on GRLIB.

- Significant FPGA resource usage. A system comprising the LEON3 SPARC processor with a 2-way set-associative 16kB cache and no memory management unit (MMU), the DDR SDRAM controller, a RS232 serial port, and an Ethernet 10/100 MAC uses 13264 FPGA look-up tables (LUT). They map to 79% of the Virtex-4 XC4VLX25 FPGA. We have carried out the test with the Xst synthesizer, Xilinx ISE 11.3, and GRLIB 1.0.21-b3957 (GPL release) using the default provided synthesis scripts. This undermines the possibility of adding hardware acceleration cores. In [22], a significant resource usage was also reported for an older version of LEON.

- Relatively low clock frequency. With the same parameters as above, the maximum clock frequency is 84MHz.

Because of these reasons, GRLIB was not retained.

ORPSoC (OpenRISC)

ORPSoC is based on the OpenRISC [26] processor core, which is the flagship product of OpenCores, a community of developers of open source system-on-chips. ORPSoC is essentially maintained by ORSoC AB.

ORPSoC notably features the OpenRISC OR1200 processor core, the Wishbone [9] bus, comprehensive debugging facilities, a 16550-compatible RS232 UART, a 10/100 Mb/s Ethernet MAC and a SDRAM controller.

Unfortunately, ORPSoC is resource-inefficient and buggy. The OpenRISC implementation is not well optimized for synthesis. We carried out tests on the August 17, 2009 OpenRISC release. Still using the XC4VLX25 FPGA as target, synthesis with Xst and Xilinx ISE 11.4 yields an utilization of 5077 LUTs for the CPU core only (using the default FPGA configuration: no caches, no MMU, multiplier, and with the implementation of the RAMs using the RAMB16 elements of the FPGA selected), running at approximately 100MHz. A similar resource usage is reported in [22]. The synthesis report shows asynchronous control signals where there should not be (such as on the output of the program counter), which can be an indication of poor quality of the design. Other IP cores comprising ORPSoC have similar issues (we tested the 16550 UART and the Ethernet MAC). Finally, the provided SDRAM controller only supports the low-bandwidth 16-bit single data rate option, has a high latency due to the extensive use of clock domain transfer FIFOs, does not support pipelined transfers and has a poorly written code.

OpenRISC and ORPSoC therefore do not seem to be a good platform for the performance-demanding and resource-constrained video synthesis application.

LatticeMico32 System

This product [30] from the FPGA vendor Lattice Semiconductor is comparable to Microblaze [34] and Nios II [4] from its competitors, respectively Xilinx and Altera.

Like its competing products, LatticeMico32 System features a broad library of light, fast and FPGA-optimized SoC cores.

One interesting move made by Lattice Semiconductor is that parts of the LatticeMico32 System are released under an open source license, and most notably the custom LatticeMico32 microprocessor core. LatticeMico32 System is also based upon the Wishbone [9] bus, whose specification is free of charge and freely distributable.

While it is perhaps technically possible to build Milkymist on top of the LatticeMico32 System, there are licensing issues concerning most notably the DDR SDRAM controller which is proprietary.

However, the LatticeMico32 microprocessor core is interesting. Synthesized for the XC4VLX25 with the 2-way set-associative caches, the barrel shifter, the hardware divider and the hardware multiplier enabled, it occupies only about 2400 4-LUTs and runs at more than 100MHz.

This microprocessor core has been retained for use in Milkymist, as described in chapter 7.

Microblaze and Nios II

Even though we are not interested in proprietary designs, we still give a brief overview of the resource usage of Microblaze and Nios II systems as a comparison.

Microblaze. In [22], the Microblaze core is reported to use approximately 2400 LUTs, like LatticeMico32. The platform builder GUI in Xilinx ISE 12.1 also limits the frequency of Microblaze systems to 100MHz when targeting the Virtex-4 family. Thus, Microblaze is close to LatticeMico32 regarding area and frequency.

Nios II. According to an Altera report [3], Nios II/f uses 1600 Cyclone II LEs. A LE is mainly comprised of a 4-LUT and a register, which is comparable to the Virtex-4 architecture on which LatticeMico32 was tested. Thus, it seems that the Nios II core would be approximately two thirds of the area of LatticeMico32.

Some differences can be noted between the LatticeMico32 configuration and the Nios II/f configuration used in the Altera report:

- Caches are direct-mapped and 512 bytes (each).
- There is no multiplier.
- Nios II/f uses a dynamic branch predictor, while LatticeMico32 uses a static branch predictor.

- The report does not say if the optional hardware divider, multiplier and shifter (that were enabled in LatticeMico32) were selected.

The Nios II is also reported to run at 140 MHz with this configuration and UART, JTAG UART, SDR SDRAM controller and timer peripherals. This is very fast, but cannot be compared to the LatticeMico32 results on Virtex-4 for two reasons:

- Routing resources and logic delays for the two FPGA families are different.
- It is possible that Altera hand-tuned the Nios II processor to their FPGA technology.

2.3 DRAM technology

DRAM is by far today's dominant memory technology, often being the only affordable solution when relatively large densities (typically more than a few megabytes) are required. Unfortunately, DRAMs are not straightforward devices and we need preliminary knowledge specific to this technology in order to understand the choices discussed in chapter 3. Indeed, in order to reduce system costs, the intelligence has been moved away from the memory chips and into the memory controller [2], leaving the controller designer with the task of dealing with the low-level details of the DRAM technology.

We will therefore explain how the SDRAM (synchronous DRAM) technology works. These principles are the same for the original single data rate (SDR) SDRAM, and for the subsequent double data rate DDR, DDR2 and DDR3 memories. In all that follows, we suppose that the logic level 0 is represented by a voltage of 0 volts, and a logic level 1 is represented by a positive voltage H.

A DRAM memory bank (figure 2.6) is organized as a two dimensional array of cells. Each cell is comprised of a transistor connected to a capacitor. A cell stores one bit of information, indicated by the presence or not of a charge in the capacitor. The transistor acts as a switch that connects the capacitor to the *bit line* (vertical lines) when the *word line* (horizontal lines) its gate is connected to carries a high logic level.

A decoder translates the row address presented to the DRAM device and activates one of the word lines, according to the address.

Each bit line is connected to a *sense amplifier*, which is a positive feedback device that, when switched on, turns any voltage X on the bit line between 0 and H into 0 (if $X < \frac{H}{2}$) or H (if $X > \frac{H}{2}$). The set of sense amplifiers is called the *page buffer*.

Accesses to a SDRAM bank are made as follows:

1. We assume the SDRAM is in the *precharged* (idle) state. In this state, no word line is active, the sense amplifiers are turned off and all the bit lines are held at a voltage of $\frac{H}{2}$.
2. The SDRAM controller presents the row address and issues an ACTIVATE command. In response to this command, the SDRAM device enables the row

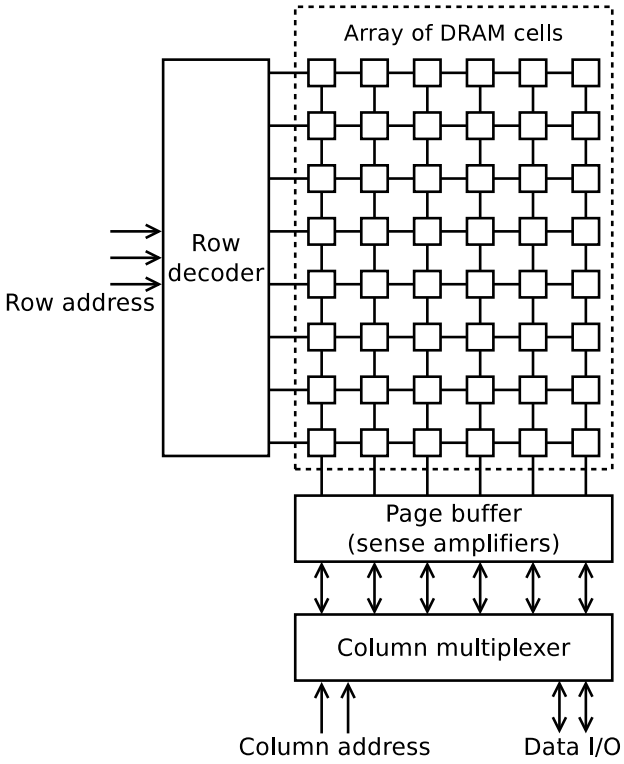

Figure 2.6. Block diagram of a DRAM memory bank.

decoder and one of the word lines is asserted. This has the effect of connecting all the capacitors of the DRAM cells in the row to their respective bit lines. A transfer of electric charge occurs between the "parasitic" capacitors of the word lines (which were charged at a voltage of $\frac{H}{2}$) and the DRAM cell capacitors, which were either discharged (at 0 volts) or charged at a voltage of H. This causes a small change ϵ in the potential of the bit line, which becomes $\frac{H}{2} - \epsilon$ or $\frac{H}{2} + \epsilon$ (depending on the charge initially stored in the DRAM cell capacitor). Then, the SDRAM device turns on all the sense amplifiers of the bank. On each bit line, the positive feedback takes over and amplifies the voltage difference ϵ until the level of the bit line reaches 0 or H. The ACTIVATE command is now completed and the row is said to be *opened*. The DDR SDRAM chips used in the project (on the Xilinx ML401 board) take 20ns to complete these operations.

3. Once a row has been opened, the controller can present the column address and issue READ and WRITE commands to transfer data. Reading is done by simply measuring the voltages on the bit lines, and writing can be achieved by forcing the bit lines to a particular level. There is a delay, called the CAS[1] latency, between a READ command being sent and the data being returned by the device. This delay is of 20ns with the chips used in the project. However, read operations are pipelined, which means that a new READ command can be sent while the previous one is still transferring data. With proper scheduling, a full utilization of the available I/O bandwidth can be achieved.

4. Before accessing another row, the memory controller must disconnect the opened row from the bit lines and go back into the precharged state. It does so by issuing a PRECHARGE command to the device. The device takes some time to process the command (during which the bank cannot be accessed), which is 20ns with the chips used in the project.

From this principle of operation, it becomes apparent that a performance-oriented controller should try to make several transfers in the same row before opening another one, in order to reduce the time wasted to switching rows.

2.3.1 Multiple banks

SDRAM memory chips contain multiple DRAM banks internally, which share the I/O, command and address pins. Additional bank address pins select the bank to send commands to.

Having multiple banks brings two advantages:

- Being able to execute several commands simultaneously (assuming there is no resource conflict for the pins). For example, one bank can be activating one row while another bank is transferring data.

[1] CAS stands for Column Address Strobe, which is the name of the DRAM chip pin that the controller asserts at this stage.

- Having several rows open (one per bank), which can reduce the number of required row switches and thus improve performance.

The controller is responsible for managing the banks, and mapping absolute memory addresses to particular banks. Appropriate bank mapping can improve performance [29].

Standard DDR SDRAM chips come with four internal banks.

2.3.2 Refreshing

Because the DRAM capacitors are not perfect, they gradually lose their charge over time, which results in data corruption.

The solution is to periodically recharge the capacitors, which is done by opening the rows one by one. SDRAM chips provide an AUTO REFRESH command which opens and closes one row in all banks (and increments an internal counter so that the next AUTO REFRESH command will target another row), but it is the responsibility of the controller to issue it. Furthermore, the controller must precharge all banks before a refresh.

With the memory chips used in the project, a refresh must be made every 7.8μs and takes in the worst case $20 + 80 + 4 \cdot 20 = 180$ns (precharge time[2] + refresh time + activation time for each bank), so it has a small impact on the memory bandwidth (about 2%).

2.4 Texture mapping

Texture mapping is a common computer graphics operation found in accelerated 3D APIs like OpenGL and DirectX. It is typically used to draw textured 3D polygons on the screen. It can also distort an image (see figure 2.7 for an example), and MilkDrop uses it for this purpose.

With common GPUs, texture mapping is performed on triangles (and more complex polygons are broke down into a series of triangles). The inputs to the algorithm are the 2D (possibly projected from the original 3D coordinates) positions of the three vertices of the triangle to be filled, and the 2D texture coordinates for these three vertices.

The algorithm then draws a textured triangle pixel by pixel, by interpolating linearly the texture coordinates of the vertices for each pixel and then copying the texture pixel (*texel*) at these coordinates.

Image processing operations like zooming, rotating or scaling can be implemented with texture mapping, by simply changing the vertices' positions or the textures coordinates at each vertex.

More often than not, the results of the linear interpolation are not integer, which means that the texture should be sampled between four adjacent pixels (figure 2.8). In this case, for a better rendering, the four pixels should be read and their colors

[2] All banks can be precharged at the same time with a single command.

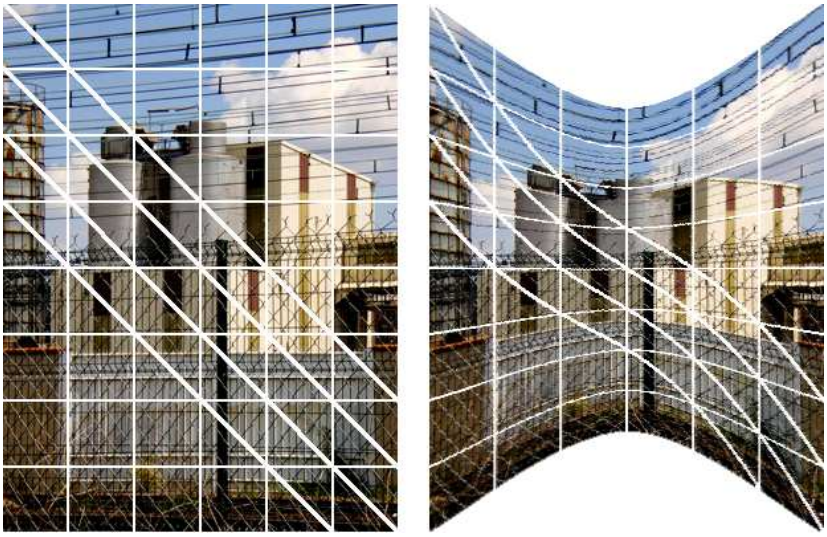

Figure 2.7. Example of distorted picture.

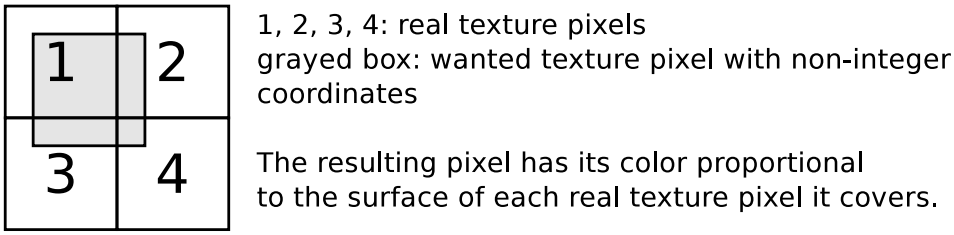

Figure 2.8. Principle of bilinear texture filtering.

should be averaged (with different weights depending on the fractional parts). This process is called *bilinear filtering* and is required to obtain a good rendering of MilkDrop presets (see figures 2.9 and 2.10).

In MilkDrop (and Milkymist), a special case of the texture mapping is used, as the only purpose is to distort a 2D image. The target surface is always a rectangle that covers the destination picture, on which the vertices are distributed evenly as a mesh which is always kept the same regardless of the applied distortion. The distortion is defined by altering the texture coordinates at each vertex.

Texture mapping, especially when bilinear filtering is desired, is a very compute intensive process, as explained in chapter 5. A custom hardware accelerator has been developed, whose details are also covered in this chapter.

2.5 Organization

According to this background, we can derive the following project guidelines:

- develop a fast, resource-efficient and FPGA-optimized system-on-chip

Figure 2.9. Rendering with bilinear filtering enabled.

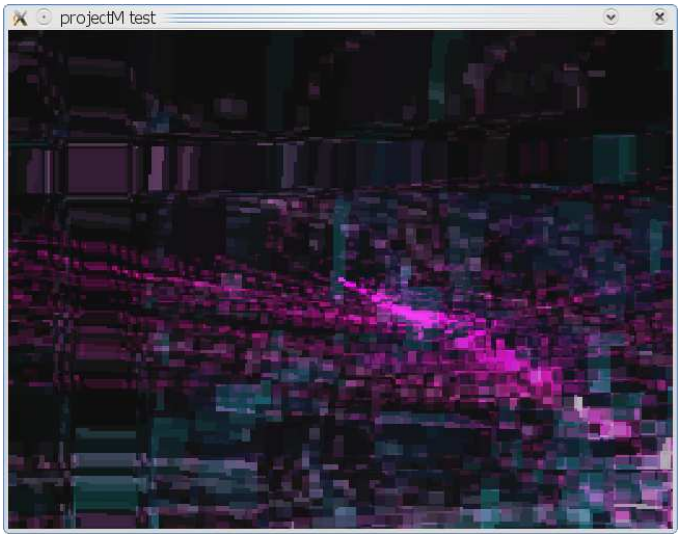

Figure 2.10. Rendering with bilinear filtering disabled (the nearest texel is used).

- develop an efficient memory subsystem
- reuse a light-weight soft-core CPU
- partition carefully the tasks between hardware and software
- develop custom hardware accelerators

The proposed solution is outlined in figure 2.11. Not all the blocks are ready at the time of this writing, nor all of them are within the scope of this Master's thesis, which focuses on computer architecture.

More specifically, the following components are not developed yet:

- microSD controller (the current prototype use a CF card through Xilinx SystemACE)
- USB controller
- Video input
- IR receiver
- MIDI controller
- DMX512 controller

Hardware accelerators have been developed for the computation of vertices positions (PFPU) and for texture mapping (TMU), which have been found to be the most compute-intensive parts of the process. They are discussed in detail in chapters 6 and 5, respectively.

Graphics processing also requires a significant amount of memory bandwidth, which is discussed in chapter 3.

Chapter 4 describes the on-chip interconnect used to make the various blocks communicate with one another.

Finally, chapter 7 deals with the software execution environment and how the software is architected to obtain a good performances from the hardware.

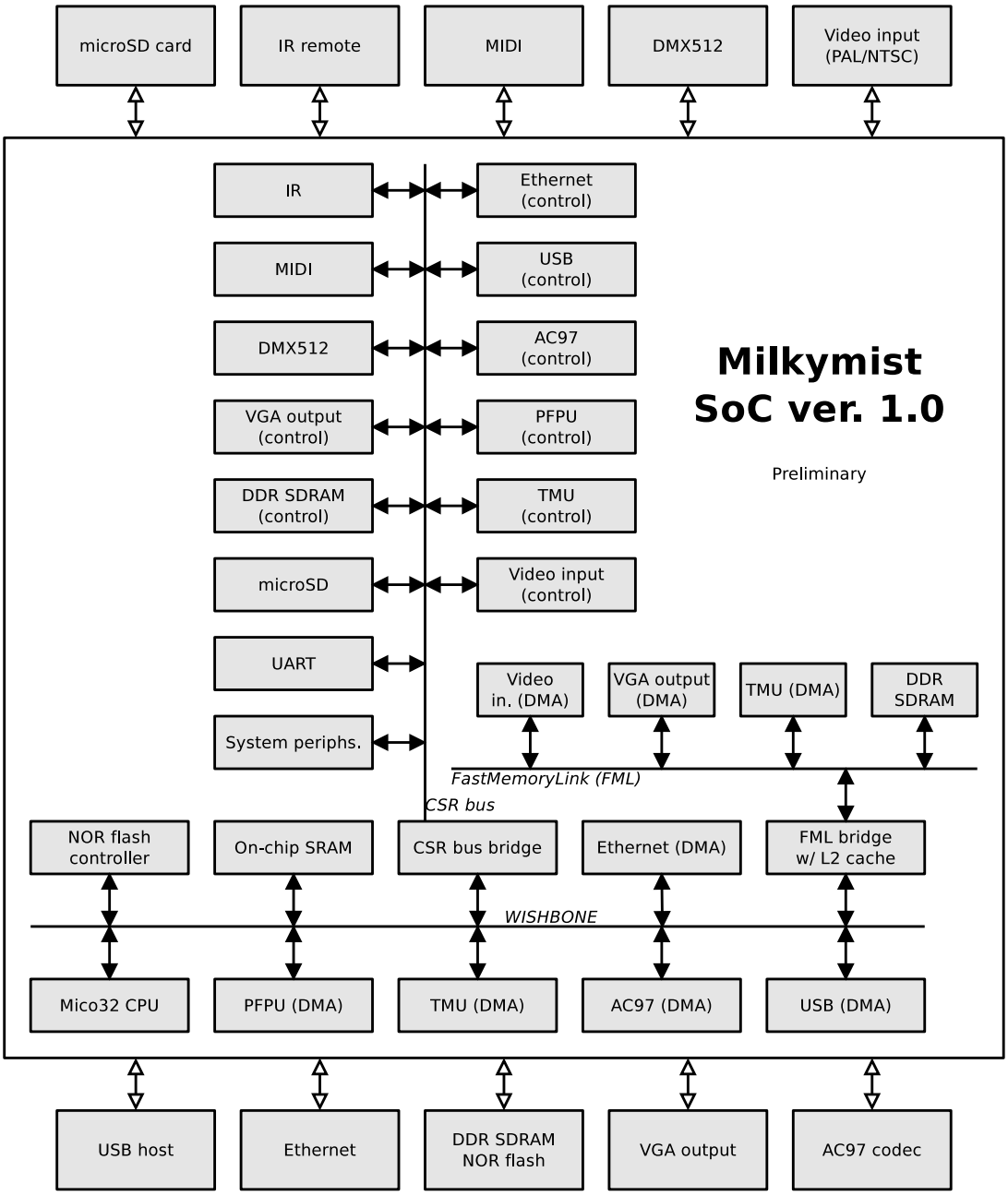

Figure 2.11. SoC block diagram.

Chapter 3

Memory subsystem

3.1 Attacking the memory wall

A recurrent point in many modern computer systems is the memory performance problem. The term *memory wall* was coined [33] to refer to the growing disparity of performance between logic such as CPUs and off-chip memories. While microprocessor performance has been improving at a rate of 60 percent per year, the access time to DRAM has been improving at less than 10 percent per year [27].

Memory performance is measured with two metrics:

- *bandwidth*, which is the amount of data that the memory system can transfer during a given period of time.

- *latency*, which is the amount of time that the memory system spends between the issue of a memory read or write request and its completion.

A memory system can have both high bandwidth and latency. If the logic making the memory accesses is able to issue requests in a pipelined fashion, sending a new request without waiting for the previous one to complete, high latency will not have an impact on bandwidth.

Latency and bandwidth are however linked in practice. Decreasing the latency also increases the bandwidth in many cases, because latency blocks sequential processes and prevents them from utilizing the full available bandwidth.

High-end processors for servers and workstations have a good ability to cope with relatively high memory latency, because techniques such as out-of-order execution and hardware multi-threading enable the processor to issue new instructions even though one is blocking on a memory access.

Some SDRAM controllers do a lot to optimize bandwidth but have little focus on latency. Bandwidth-optimizing techniques include:

- reordering memory transactions to maximize the page mode hit rate.

- grouping reads and writes together to reduce write recovery times. Along with the above technique, this has a detrimental impact on latency because of the delays incurred by the additional logic in the address data path.

- running the system and the SDRAM in asynchronous clock domains in order to be able to run the SDRAM at its maximum allowable clock frequency. This requires the use of synchronizers or FIFOs, which have a high latency.

- configuring the SDRAM at high CAS latencies in order to increase its maximum allowable clock frequency. This trend is best illustrated by the advent of DDR2 and DDR3 memories whose key innovation is to run their internal DRAM core at a sub-multiple of the I/O frequency with a wide data bus which is then serialized on the I/O pins. Since the internal DRAM core has a latency comparable to that of the earlier SDR and DDR technologies, the number of CAS latency cycles relative to the I/O clock is also multiplied.

An extreme example of these memory controller bandwidth optimizations is the MemMax® DRAM scheduler [17]. This unit sits on top of an already existing memory controller (which already has its own latency), adding seven stages of complex and high-latency pipelining that produces a good - but compute-intensive - DRAM schedule. The actual efficiency of this system has been questioned [15] because of that significant increase in latency.

3.2 Another approach

The out-of-order execution and hardware multi-threading processor optimizations discussed above that cope with high memory latency are complex and impractical in the context of small and cheap embedded systems, especially those targeted at FPGA implementations. For example, FPGA implementations of the OpenSPARC [23] processor, which employs such optimizations, typically require an expensive high-end Xilinx XUPV5 board whose Virtex-5 FPGA alone costs roughly 13000 SEK.

Milkymist therefore uses simple in-order execution schemes in its CPU and in its accelerators, and strives to improve performance by focusing on reducing the memory latency.

The memory system features that improve latency (but also bandwidth) are discussed below.

3.3 Memory system features

3.3.1 Single SDRAM and system clock domain

The typical operating frequency of early SDR and DDR SDRAM (technologies that are prior to DDR2 and do not have a clock divider for the internal DRAM core) is close to the 100MHz frequency at which the FPGA is able to meet timing for

the complete SoC. Thus, it was decided to run the DRAM and the system synchronously in order to remove the need for any clock domain transfer logic and reduce latency. The SDRAM I/O registers are clocked by the system clock, and timing of the SDRAM interface is met through the use of calibrated on-chip delay elements and delay-locked-loops (DLLs) to generate the off-chip SDRAM clock and the data strobes.

3.3.2 Page mode control algorithm

The Milkymist memory controller takes the so-called *page mode gamble*: after an access, the DRAM row is left open in the hope that the next transaction to the memory bank will occur within the same row. If the memory controller is right, the read or write command can be immediately registered into the SDRAM, and only the CAS or write latency is incurred. If the memory controller is wrong, it must first precharge the DRAM bank and open the correct row, causing extra delays.

Thus, if the memory controller is often wrong, taking the page mode gamble will actually impact performance negatively. However, a study has shown [29] that, with typical memory timings, the point at which the gamble pays off is for a page hit probability of 0.375 only, attainable with many practical memory access patterns.

Page hit probability is improved by the ability of the Milkymist memory controller to track open rows independently in each of the four memory banks that commercial SDRAM chips are equipped with.

This optimization positively affects both latency and bandwidth.

3.3.3 Burst accesses

All memory accesses are made using bursts, i.e. when an access for a word is made, the following words are also read or written. Burst mode is a feature of the SDRAM chips: only one read of write command is sent to them, and several words are transferred on subsequent clock cycles.

Using bursts frees the bus and DRAM control signals while other words are transferred, allowing the issue of new commands overlapping the data phase of the previous transaction.

Burst access is a form of prefetching that improves latency. It is only efficient when the prefetched data can be used by the requesting bus master. In the Milkymist system-on-chip, this is often the case:

- The CPU core has caches which access memory by complete cache lines. Thus, if the cache line length is a multiple of the burst length, the bursts can be easily fully memorized.

- The video frame buffer repeatedly reads the same block of data in a sequential manner, and can easily make full use of the prefetched data assuming that is has sufficient on-chip buffer space.

- The texture mapping unit also has a cache and a write buffer which work well with burst accesses. This is discussed in Chapter 5.

3.3.4 Burst reordering

This feature enables the use of the critical-word-first scheme in caches, reducing the overall memory latency.

When a request is issued at an address which is not a multiple of the burst length, the order of the words in the burst is changed so that the first word that comes out is the very word that is at the requested memory address. The prefetch address is then incremented and wraps to stay within the same burst.

For example, assuming a burst length of 4:

- a request at address 0 fetches words 0, 1, 2 and 3 (in this order)
- a request at address 2 fetches words 2, 3, 0 and 1 (in this order)

3.3.5 Pipelining

The memory bus of Milkymist [8] is pipelined. During the transfer of the prefetched (burst) data, a new request can be issued. This is illustrated for a read request by the table below:

Address	A1	A1	A1	A2	A2	A2	A2
Data	–	–	M(A1)	M(A1+1)	M(A1+2)	M(A1+3)	M(A2)

Address (cont.)	–	–	–
Data (cont.)	M(A2+1)	M(A2+2)	M(A2+3)

Together with burst access, this helps achieving high performance: the memory controller can hide DRAM latencies and row switch delays by issuing the requests to the DRAM in advance, while the previous transaction is still transferring data.

3.4 Practical implementation

The Milkymist SoC uses 32-bit DDR SDRAM, configured to its maximum burst length of 8. Since the DDR SDRAM transfers two words per clock cycles (one on each edge), this is turned by the I/O registers into bursts of four 64-bit words synchronous to the system clock.

The memory is run at 100MHz, yielding a peak theoretical bandwidth of 6.4Gb/s, which is more than enough for the intended video synthesis application (table 3.1). This bandwidth is however never attained: events such as switching DRAM rows which takes significant time and, to a lesser extent, DRAM refreshes introduce dead times on the data bus. We will see in section 3.5 that such an oversizing of the off-chip memory is needed if we want to keep the memory system simple.

Task	Required bandwidth
VGA frame buffer, 1024x768, 75Hz, 16bpp	950Mb/s
Distortion: texture mapping, 512x512 to 512x512, 30fps, 16bpp	250Mb/s
Live video: texture mapping, 720x576 to 512x512 with transparency, 30fps, 16bpp	300Mb/s
Scaling: texture mapping, 512x512 to 1024x768, 30fps, 16bpp	500Mb/s
Video echo: texture mapping, 512x512 to 1024x768 with transparency, 30fps, 16bpp	900Mb/s
NTSC input, 720x576, 30fps, 16bpp	200Mb/s
Software and misc.	200Mb/s
Total	**3.3Gb/s**

Table 3.1. Estimate of the memory bandwidth consumption.

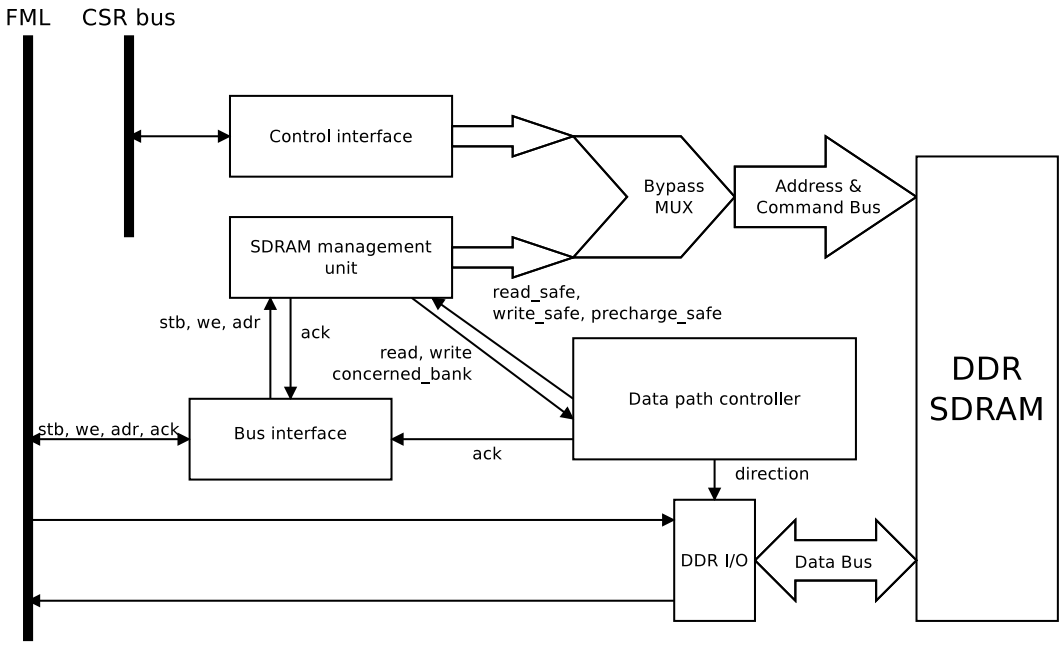

Figure 3.1. Block diagram of the HPDMC architecture.

The architecture of the memory controller, called HPDMC (for "High Performance Dynamic Memory Controller"), is outlined in figure 3.1.

The control interface is used by the system to configure the controller, and also to issue the start-up sequence to the SDRAM. Indeed, SDRAM chips require a sophisticated sequence of commands upon power-up. In many memory controller designs, a hardware finite state machine is used to issue this command sequence. In order to save hardware resources, the system used here leaves this task to the

software, and, for this purpose, includes a "bypass MUX" that routes directly a configuration and status register of HPDMC to the SDRAM command and address pins. Once the SoC has run a software routine that sends the correct initialization sequence to the SDRAM, it switches permanently the bypass MUX to the "SDRAM management unit" and can use off-chip memory normally.

The SDRAM management unit is a finite state machine that translates the two high-level memory commands "read burst at address" and "write burst at address" into a series of lower-level commands understandable by the SDRAM chips (precharge bank, select row, read from row, etc.). The management unit is responsible for keeping track of the open rows, detecting page hits, switching rows, and issuing periodic DRAM refresh cycles.

The management unit is connected to the "data path controller", that follows the activities performed by the management unit in order to decide the direction of the bidirectional I/O pins (they should be set as outputs for writes and as input for reads). The data path controller is also responsible for sending signals to the management unit that indicate if it is safe to perform certain low-level operations. For example, the `read_safe` signal goes low immediately after a read command is issued, because if another one were sent immediately after, the two resulting bursts would overlap in time and this could not work because there is only one set of data pins. Eventually, the data path controller takes into account the SDRAM write and read latencies to generate an acknowledgement signal when the data is actually there (or needs to be sent to the SDRAM) after a "read row" or "write row" command has been sent to the SDRAM.

Finally, the bus interface is a piece of glue logic that connects the SoC pipelined memory bus (FML) to the data path controller and the management unit.

HPDMC has been implemented in Verilog HDL, tested and debugged in RTL simulation using a DDR SDRAM Verilog model from Micron, integrated into the SoC, synthesized into FPGA technology, and eventually calibrated and tested by software routines running on the actual hardware.

This design of memory controller, specifically crafted for the Milkymist project and released under the GNU GPL license on the internet, has been picked up by the NASA for a software defined radio project and may be put up on board the international space station in 2011. Gregory Taylor, Electronics Engineer at the NASA Jet Propulsion Laboratory, wrote:

While searching for a suitable SDRAM controller for the Jet Propulsion Laboratory's Software-Defined Radio on board NASA's CoNNeCT experiment, I found Sébastien's HPDMC SDRAM controller on OpenCores.org. We needed a controller that was both high performance and well documented. Though the original HPDMC controller was designed for DDR SDRAM with a 32-bit bus, Sébastien clearly explained the modifications necessary to adapt the controller to our Single Data Rate, 40-bit wide SDRAM chip. I found the code to be well documented and easy to follow. The performance has met our requirements and the FPGA size requirement is small.

The Communication Navigation and Networking Reconfigurable Testbed (CoNNeCT) experiment to be installed on board the ISS is designed for the next generation

of SDRs conforming to the Space Telecommunications Radio Systems (STRS) open architecture standard. The HPDMC controller will likely find its way into one or more loadable waveform payloads in the JPL SDR, and perhaps be used in other NASA projects as well. It may eventually find its way into deep space.

3.5 Performance measurement

3.5.1 Introduction

We wanted to validate and characterize the memory system performance (actual latency and bandwidth) and get an upper bound of of its ability to sustain loads, by extrapolating the maximum bandwidth one could get assuming the memory access time remains constant.

Since the memory performance depends on the particular access pattern that the system makes (because of the controller taking the page mode gamble, we wanted to take the measurements on the real system while it is rendering video effects in order to get an accurate result.

3.5.2 Method

A logic core has been added to the SoC that snoops on the memory bus activity in order to report the average latency and bandwidth.

That logic core exploits properties of the FastMemoryLink signaling in order to reduce its complexity to two counters that measure, for a given time period, the number of cycles during which the strobe and acknowledgement signals are active. Several parameters can then be computed:

- the **net bandwidth** carried by the link (based on the amount of data that the link has actually transferred)
- the **average memory access time**, which is the time, in cycles, between the request being made to the memory controller and the first word of data being transferred.
- the **bus occupancy** which is the percentage of time during which the link was busy and therefore unavailable for a new request.

Every FastMemoryLink transaction begins with the assertion of the strobe signal. Then, after one or more wait cycles, the memory controller asserts the acknowledgement signal together with the first word of data being transferred. The next cycle, the strobe signal is de-asserted (unless a new transaction begins) while the next word in the burst is being transferred. A new transaction can start with the assertion of the strobe signal even if a burst is already going on (pipelining). See figure 3.2 for an example.

In the equations that follow, these symbols are used:

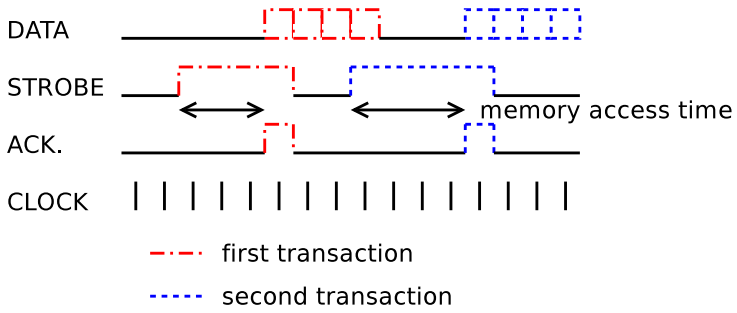

Figure 3.2. FML transactions.

- f is the system clock frequency in Hz.
- T is the time during which the counters have been enabled.
- w is the width of a FML word in bits.
- n is the FML burst length.
- S is the number of cycles during which the strobe signal was active.
- A is the number of cycles during which the acknowledgement signal was active.

Net bandwidth. By counting the number of cycles for which the acknowledgement signal was active, one gets the number of transactions. Since each transaction carries exactly a burst of data, which is $w \cdot n$ bits in size, the volume of data transferred is given by $w \cdot n \cdot A$. Thus, one can derive the net bandwidth as:

$$\beta = \frac{w \cdot n \cdot A}{T} \tag{3.1}$$

Average memory access time. On the bus, a master is waiting when the strobe signal is asserted but the acknowledgement signal is not. Therefore, the total number of wait cycles is given by $S - A$. The average memory access time can thus be computed as:

$$\Delta = \frac{S - A}{A} \tag{3.2}$$

The average memory access time can be used to derive an upper bound on the maximum bandwidth that the memory system can handle. Indeed, FML is a pipelined bus which supports only one outstanding (waiting) transaction, so the case that uses the most bandwidth for a given memory access time is when the strobe signal is always asserted (figure 3.3) so that a new transaction begins as soon as the first word of the previous transaction is transferred.

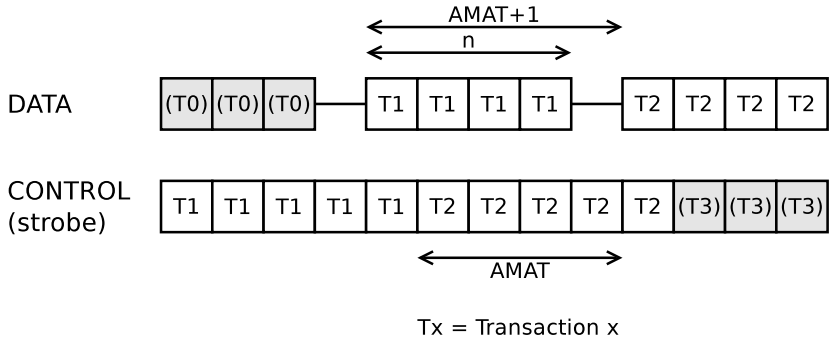

Figure 3.3. Maximum utilization of a FML bus.

Therefore, only a fraction α of the peak bandwidth $f \cdot w$ can be used at most, and we have:

$$\alpha = \max(1, \frac{n}{\Delta + 1}) \tag{3.3}$$

The maximum bandwidth is:

$$\beta_{max} = \alpha \cdot f \cdot w \tag{3.4}$$

Bus occupancy. The bus is busy when the strobe signal is asserted. The bus occupancy is therefore given by:

$$\epsilon = \frac{S}{T \cdot f} \tag{3.5}$$

By using this method, a very simple piece of hardware added to the system can yield to the retrieval of interesting information about the performance of the memory system.

3.5.3 Results

Results are summarized in table 3.2. The first line corresponds to a system running the demonstration firmware with the video output enabled at the standard VGA mode of 640x480 at 60Hz (therefore continuously scanning the screen with data from system memory), but not rendering a preset. The other lines represent the results while the demonstration firmware is rendering different MilkDrop presets, still at the same video resolution.

It is difficult to compare these results to those of other memory controllers as they are usually not published (or not measured at all).

However, two conclusions can be drawn:

- there are enough occupancy and bandwidth margins for the system to operate at higher resolutions and/or color depths than 640x480 and 16 bits per pixel. The 3.3 Gb/s bandwidth requirement that was estimated in section 3.4 seems attainable, although challenging.

Patch	β	ϵ	Δ	α	β_{max}
Idle	292	7 %	5.51	61 %	3932
Geiss - Bright Fiber Matrix 1	990	28 %	6.37	54 %	3474
Geiss - Swirlie 3	1080	32 %	6.71	52 %	3320
Geiss - Spacedust	1021	29 %	6.47	54 %	3427
Illusion & Rovastar - Snowflake Delight	1399	39 %	6.28	55 %	3516
Rovastar & Idiot24-7 - Balk Acid	1427	41 %	6.38	54 %	3469

Table 3.2. Memory performance in different conditions (Milkymist 0.5.1). Bandwidths are in Mb/s.

- to go further, an "out-of-order" memory controller can be envisioned. Such a controller would have a split transaction bus (allowing a larger number of outstanding transactions, thus minimizing the impact that latency has on bandwidth) and would be able to reorder pending memory transactions to maximize the page hit rate.

Chapter 4

SoC interconnect

This chapter explains how the different interconnect busses work, what their features are, why they are there, and how they are communicate with each other.

The general SoC block diagram and its interconnect is outlined in figure 2.11.

4.1 General SoC interconnect: the Wishbone bus

Wishbone [9] is a general purpose royalty-free SoC bus with open specifications, advocated by the maintainers of the OpenCores.org website.

Wishbone is a synchronous sequential bus with support for variable latency (wait states) through the use of an acknowledgement signal that marks the end of the transaction. Burst modes (automatic transfer of consecutive words) are supported and are configurable on a per-transaction basis (i.e. bursts of arbitrary lengths and single-word transactions can be freely mixed on the same bus). However, there is no pipelining.

Wishbone is used around the SoC's LatticeMico32 CPU core and for simple DMA masters which have modest requirements of bandwidth and of volume of transferred data. As explained in Section 4.4, connecting DMA masters that transfer small amounts of data (which can fit in the L2 cache) to the same bus as the CPU simplifies dealing with cache coherency issues.

The data width used for the Wishbone bus is 32, yielding a peak bandwidth of 3.2Gb/s when the system is running at 100MHz.

4.2 Configuration and Status Registers: the CSR bus

Milkymist uses memory-mapped I/O through configuration and status registers.

If these registers were directly accessed by the Wishbone CPU bus, two problems would arise:

- Connecting all peripherals on the same Wishbone bus involves large multiplexers and high fanout signals, posing routing and timing problems.

- Wishbone requires the generation of an acknowledgement signal by each slave core. This signal is useful in many cases, as it supports peripherals with a variable latency. However, configuration and status register files are usually implemented with actual registers (flip flops) or SRAM, which can always be accessed in one clock cycle. Thus, there is no need for variable latency and the acknowledgement signal. Keeping this signal for the configuration and status registers wastes hardware resources and development time.

To alleviate these problems, the CSR bus has been developed [7] and used in the system through a bus bridge.

The CSR bus is a simpler bus than Wishbone, where all transfers are done in one cycle. It has an interface similar to that of synchronous SRAM, consisting only of address, data in, data out and write enable pins and clocked by the system clock.

A bridge connects the CSR bus to the CPU Wishbone bus, to allow transparent memory-mapped access to the configuration and status registers by the software. This bridge includes registers for all the signals crossing the two busses, relaxing the timing constraints.

4.3 High-throughput memory access bus: the FML bus

FastMemoryLink (FML) [8] was co-designed with HPDMC (the memory controller) as a on-chip bus tailored to access SDRAM memories at high speed while keeping the memory controller simple. Its key features are listed below.

4.3.1 Variable latency

SDRAM latency varies a lot depending on the state of the SDRAM at the time the request is issued on the bus. It depends on whether the SDRAM was in the middle of a refresh cycle, whether the bank needs to be precharged, and whether a new row needs to be activated. Therefore, FML provides support for a variable number of wait states, defined by the memory controller, through the use of an acknowledgement signal similar to that of Wishbone.

4.3.2 Burst only

SDRAM is best accessed in burst mode (see subsection 3.3.3).

However, enabling or configuring burst mode is a relatively lengthy and complex operation, requiring a reload of the SDRAM mode register which takes several cycles. Furthermore, supporting multiple burst lengths makes the scheduling of the transfers more complex to avoid "overlapping" transfers that would create conflicts at the data pins.

Therefore, in order to greatly simplify the memory controller, all transfers on FML are made using a fixed and pre-defined burst length.

4.3.3 Burst reordering

This was discussed in subsection 3.3.4.

4.3.4 Pipelining

The benefits of this feature have already been discussed in subsection 3.3.5.

Pipelined requests may come from the same core that issued the initial transfer, or from another core. The FML arbiter would then pipeline the request coming from the other core.

4.3.5 Usage

The data width used for the FML bus is 64, yielding a peak bandwidth of 6.4Gb/s when the system is running at 100MHz. This is twice the peak bandwidth of the Wishbone bus. Furthermore, this bus provides a short path to the memory controller, reducing latency and therefore potentially further increasing effective bandwidth, as discussed in Section 3.1.

Peripherals directly connected to FML are typically those which transfer large amounts of data (that would exceed the capacity of the L2 cache presented in section 4.4) and which have high bandwidth requirements (and therefore can take advantage of the bandwidth and latency improvement compared to Wishbone).

In the Milkymist SoC, they are comprised of:

- the VGA output controller, which needs to continuously scan a frame buffer up to several megabytes in size to generate the video signal.
- the (planned) video input, which writes, every second, dozens of digitized video frames weighting hundreds of kilobytes each.
- the texture mapping unit (chapter 5), which needs to deal with large textures at high speed.

4.4 Bridging Wishbone to FML

For Wishbone masters (like the CPU) to access SDRAM transparently, it is necessary to bridge the FML bus to the Wishbone bus.

FML is a burst-only bus with a fixed burst length, while with Wishbone, bursts are optional and configured on a per-transaction basis. To be efficient, the bridge must therefore be able to store data and slice it to meet the transfer size requirements of the Wishbone and FML transactions.

A traditional write-back cache with a line length equal to the FML burst length provides an elegant solution to this problem. This cache is referred to as the "L2 cache", because, from the CPU point of view, it provides a second level of cache relative to its integrated instruction and data caches.

4.5 Cache coherency

4.5.1 Coherency issues around the CPU (L1) cache

The LatticeMico32 CPU (section 7.1) uses a write-through cache without hardware coherency. Thus, the following operations must be done by the software to ensure cache coherency:

- Before reading DMA data from a peripheral using shared memory, the L1 cache should be cleared as it may hold an outdated copy of the data.
- When writing DMA data to a peripheral using shared memory on the Wishbone bus, no precaution should be taken. The CPU writes go directly to the bus, and end up in the L2 cache or the SDRAM where the peripheral will correctly retrieve them.

It is noteworthy that the CSR address space is non-cache-able, therefore no cache-related precaution should be taken when reading or writing CSRs.

4.5.2 Coherency issues around the Wishbone-FML (L2) cache

The Wishbone-FML bridge provides very limited support for cache coherency. Cache coherency issues arise because of the masters directly connected to the FML bus:

- The CPU may read a cached copy of a data that has been modified by a FML master.
- A FML master may read a value that has been modified by the CPU in the cache (dirty line) but not flushed to the SDRAM.
- A FML master may update a value in SDRAM but not in the cache. The line may then go dirty, and, when flushed, will erase the value written by the FML master.

Because cache coherency is expensive to implement in hardware, the task of managing the coherency of caches has been moved almost entirely to the software. The bridge exposes an interface for the software to invalidate cache lines, flushing them to the SDRAM if they are dirty. On the software side, device drivers should use this interface appropriately when transferring data with hardware units that use shared memory.

The only form of hardware cache coherency the system has is related to the video frame buffer. The VGA signal generator is connected directly to the SDRAM bus, because the frame buffer is continuously scanned and is too large to fit entirely in the cache.

However, it is very common that software modifies only a few pixels on the screen. If there was no hardware cache coherency at all, it would be tricky to implement a software mechanism that flushes the bridge cache at appropriate times. A solution can be to flush the cache every time a pixel or a group of pixels are written (which

can be extremely slow if only small regions of the screen are modified at a time). Another solution would be to periodically check if the frame buffer had been modified and flush the cache if it was.

Since those solutions are difficult to implement as they require a significant support from both the operating system and applications, it was chosen to make frame buffer read transactions by the VGA signal generator coherent with respect to the bridge cache. Every time the VGA signal generator fetches a burst of pixels, it first searches the bridge cache. If the data is in the cache, it is used. If not, the VGA signal generator fetches it from SDRAM (but does not replace any cache line).

This also makes it easier to write Milkymist frame buffer drivers within the frameworks of common operating systems, such as Linux (figure 7.2).

Chapter 5

Texture mapping unit

High performance texture mapping was perhaps the most challenging and interesting part of the SoC design project. This chapter begins with the design of an efficient algorithm and continues with the hardware implementation of it, and in particular how it was pipelined and how its memory references are handled.

5.1 Algorithm

5.1.1 Two-dimensional interpolation

As underlined in section 2.4, we need to interpolate linearly on a 2D polygon the X and Y texture coordinates according to known values on the vertices of the polygon.

Traditional GPUs use the triangle as the primitive polygon, because it allows them to draw any other polygon by splitting it into a series of triangles. We do not need such flexibility. For rendering MilkDrop presets, the surface to be drawn is always a mesh of rectangles whose edges are parallel to the borders of the screen (traditional GPUs draw the surface using a triangle decomposition similar to the one shown in figure 5.1). We can therefore choose, as the primitive polygon, the rectangle with edges parallel to the borders of the screen, which is much simpler to draw than arbitrary triangles.

We then split the 2D interpolation problem into 1D interpolation problems as follows:

- The X and Y texture coordinates are interpolated independently.
- First, each coordinate is interpolated on the vertical edges of the rectangles (vertical interpolation) for each integer value of the ordinate.
- For each integer value of the ordinate, the results from the vertical interpolation are interpolated again for each integer value of the abscissa (horizontal interpolation). This scans all the pixels within the rectangle.

The process is illustrated in figure 5.2.

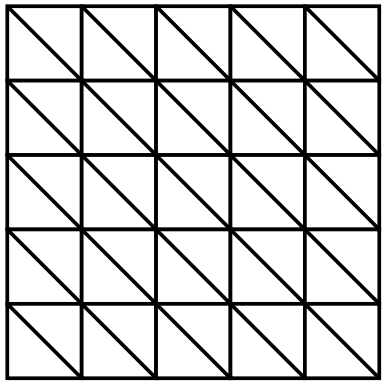

Figure 5.1. Typical decomposition into triangular primitives of the MilkDrop rendering surface.

One could obtain the same result by starting with an horizontal interpolation followed by several vertical interpolations. However, when using a linear scan frame buffer (as Milkymist does), doing it in the proposed way yields output pixels whose memory addresses are consecutive in most cases (except when going to the next ordinate), which works well with the bursty nature of SDRAM accesses (subsection 3.3.3) and the traditional organization of a cache.

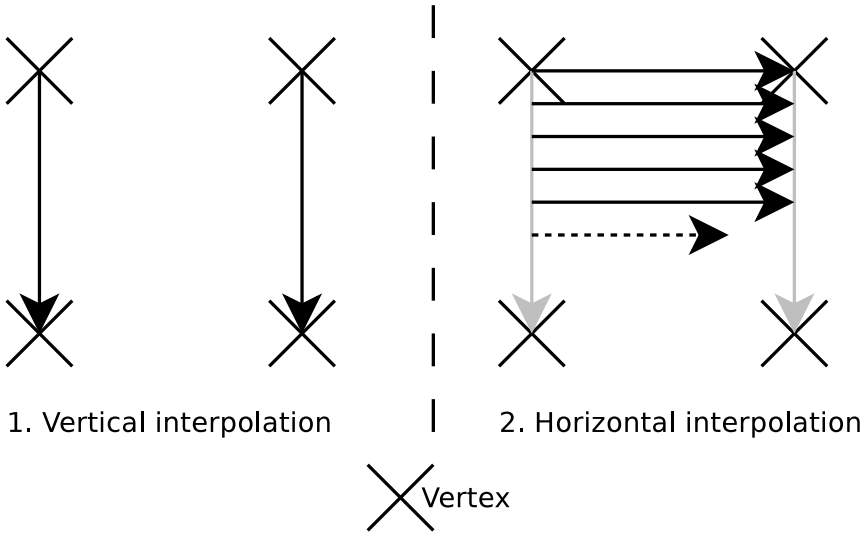

Figure 5.2. 2D linear interpolation on a rectangle.

5.1.2 One-dimensional interpolation

The problem now boils down to performing one-dimensional linear interpolations. Given two points $A(x_0, y_0)$ and $B(x_1, y_1)$ with integer coordinates, we need to compute the ordinate y of $M(x, y) \in (AB)$ for all the integer values of x between x_0 and

$Dy \leftarrow y_1 - y_0$
$Dx \leftarrow x_1 - x_0$
$Q \leftarrow Dy/Dx^a$
$R \leftarrow Dy\%Dx^b$
$x \leftarrow x_0$
$[y] \leftarrow y_0$
$e \leftarrow 0$
result(x) $\leftarrow [y]$
while $x < x_1$
    $x \leftarrow x + 1$
    $[y] \leftarrow [y] + Q$
    $e \leftarrow e + R$
    if $2 \cdot e > Dx$
        $[y] \leftarrow [y] + 1$
        $e \leftarrow e - Dx$
    end
    result(x) $\leftarrow [y]$
end

[a] / is the integer division operator
[b] % is the integer modulo operator

Figure 5.3. One-dimensional linear interpolation algorithm.

x_1. For now, we are not interested in bilinear filtering, so what we actually want is the best integer approximation $[y]$ of y so that the texture is sampled to the nearest pixel.

In figure 5.3 we propose a fast and integer-only[1] algorithm, which was inspired by Bresenham's line drawing algorithm [1]. Without loss of generality, we suppose that $x_0 \leq x_1$ (the points can be reordered if this was not the case). We also suppose that $y_0 \leq y_1$ (it is easy to modify the algorithm to handle the $y_0 > y_1$ case as well[2]).

The correctness of the algorithm lies in the fact that every time the result is being written, those conditions are verified (from which it can be derived that $[y]$ is the best integer approximation of y):

1. $|e| \leq \frac{Dx}{2}$ (which implies $|\frac{e}{Dx}| \leq \frac{1}{2}$)

2. The perfect (rational) interpolated value y is equal to $y = [y] + \frac{e}{Dx}$.

These conditions can be proven true by recursion:

1. For $x = x_0$, $e = 0$ therefore $|e| \leq \frac{Dx}{2}$. Let us now suppose that the hypothesis is true for a certain value of $x \geq x_0$, and prove that it is true for $x + 1$.

[1] And thus more suited to a resource-constrained hardware implementation.
[2] See the Verilog implementation in Milkymist (cores/tmu2/rtl/tmu2_geninterp18.v).

The instructions that affect e between two consecutive values of x are $e \leftarrow e+R$ and, if $2 \cdot e > Dx$, $e \leftarrow e - Dx$.

After the first instruction:

- if e was negative or zero, we have $-\frac{Dx}{2} \leq e < Dx < \frac{3 \cdot Dx}{2}$ (because $0 \leq R < Dx$ and the recursion hypothesis).
- if e was positive, it was inferior or equal to $\frac{Dx}{2}$ (because of the recursion hypothesis), therefore we have $0 < e < \frac{3 \cdot Dx}{2}$.

After the second instruction, if we had $e > \frac{Dx}{2}$, we'll have $e < \frac{3 \cdot Dx}{2} - Dx$. Therefore, $e \leq \frac{Dx}{2}$. □

2. We need to prove that every time the result is being written, the following equation is verified:

$$[y] + \frac{e}{Dx} = y_0 + \frac{y_1 - y_0}{x_1 - x_0} \cdot (x - x_0) \tag{5.1}$$

For $x = x_0$, $[y] + \frac{e}{Dx} = y_0$, so the equation is verified. Let us now suppose that it is verified for a certain value of $x \geq x_0$, and prove that it is true for $x + 1$.

It can be noted that the instructions within the "if" do not change the value of $[y] + \frac{e}{Dx}$. The only instructions that change the result between two consecutive values of x are $[y] \leftarrow [y]+Q$ and $e \leftarrow e+R$. Therefore, after the loop iteration, we have:

$$[y] + \frac{e}{Dx} = y_0 + \frac{y_1 - y_0}{x_1 - x_0} \cdot (x - x_0) + Q + \frac{R}{Dx} \tag{5.2}$$

$$[y] + \frac{e}{Dx} = y_0 + \frac{y_1 - y_0}{x_1 - x_0} \cdot (x - x_0) + \frac{y_1 - y_0}{x_1 - x_0} \tag{5.3}$$

$$[y] + \frac{e}{Dx} = y_0 + \frac{y_1 - y_0}{x_1 - x_0} \cdot ((x + 1) - x_0) \tag{5.4}$$

□

5.1.3 Bilinear filtering

As outlined in section 2.4, bilinear filtering is needed to obtain good rendering results.

We will therefore try to improve the previous algorithm so that we get a more precise, non-integer interpolation result. Preferably, the result should be in fixed point format so that it can be easily handled for the actual filtering stage (weighted average of adjacent pixels colors).

The first thing that comes to mind is to try to use the error value $\frac{e}{Dx}$. However, this would require an integer to fixed point division to be performed at each interpolated result (the horizontal interpolation alone would require two such operations per pixel), which is expensive.

A more elegant solution consists in multiplying all the texture coordinates by a power of 2, noted S (this is an inexpensive operation, as it can be implemented with

a bit shift). Since the interpolation process is linear, the outputs are also multiplied by S — but the precision is increased. In other words, the output of the interpolation stages comes directly in fixed point format, with $log_2(S)$ digits after the radix point.

Figure 5.4 illustrates how the bilinear filtering is done using the fixed point texture coordinates. The texture coordinates are noted X.i, X.f, Y.i and Y.f, with "i" denoting the integer part of the fixed point number (bits before the radix point) and "f" denoting the fractional part (bits after the radix point).

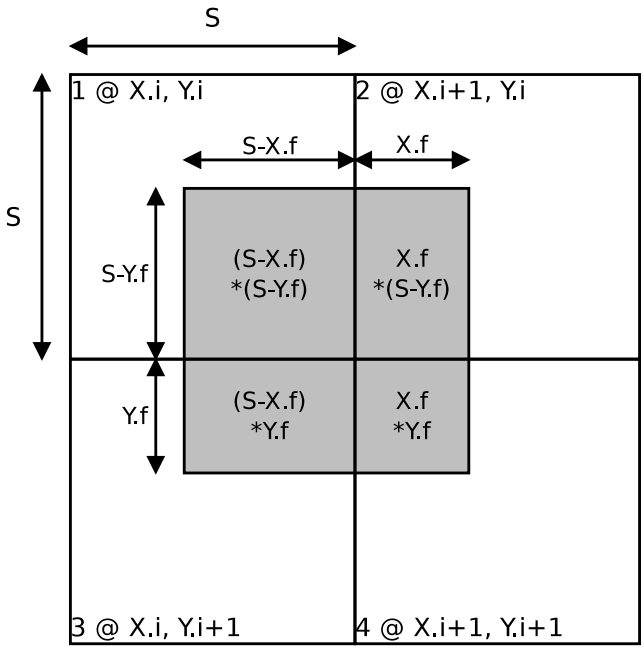

Figure 5.4. Bilinear filtering using the fixed point texture coordinates.

The weights in the average should be proportional to the surface that the texel to be sampled with non-integer coordinates (the grey box on the figure) covers in each of the real texture pixels (numbered 1 to 4 on the figure). Thus, if c_1, c_2, c_3 and c_4 are respectively the color vectors of the texture pixels 1 to 4 and c is the color vector of the result, we have:

$$c = \frac{(S - X.f)(S - Y.f) \cdot c_1 + X.f(S - Y.f) \cdot c_2 + (S - X.f)Y.f \cdot c_3 + X.fY.f \cdot c_4}{S^2} \tag{5.5}$$

Since we are working with the RGB565 color format, having more than 6 extra bits of precision would not make a difference for the filtering. Therefore, we choose $S = 2^6 = 64$.

Operation	Cost
Addition or subtraction	1 cycle
Multiplication	2 cycles
Division or modulo	32 cycles
Bit shift	1 cycle
Test ($<$, $>$, $=$, etc.)	1 cycle
Conditional jump	3 cycles
Assignment	free (which is optimistic)
Reading or writing to the frame buffer	2 cycles

Table 5.1. Estimates of the cost of common software operations.

5.2 Performance considerations

5.2.1 Context

To motivate the implementation choice of the texture mapping, we will study its execution time in the following situation:

- The size of the source (texture) and destination pictures is 512x512.
- The size of the primitive rectangles is 16x16.
- We need at least 120 runs per second. Indeed, the renderer needs to distort the image, include live video, scale it, and apply the video echo effect 30 times per second (subsection 2.1.2). We therefore have approximately 8 ms of processing time at most (which corresponds to 31 megapixels per second). This is a very optimistic estimate: since scaling, inclusion of live video and the video echo effect work with resolutions greater than 512x512, these processes are expected to take more time than the 512x512 $\rightarrow$ 512x512 distortion.
- The system clock is 100MHz.
- The $2 \cdot e > Dx$ test is always false (which is optimistic).
- We optimistically do not take into account the extra instructions needed to handle interpolations with a negative slope ($y_0 > y_1$).

For a software implementation, we use the cost estimates of table 5.1.

5.2.2 Execution time of the interpolation algorithm

For each 1D interpolation with n steps, we need the amount of time detailed in table 5.2. The steps are in the same order as in figure 5.3.

Operation	**Cycles**
2 subtractions	2
Division	32
Modulo	32
Test	n
Conditional jump	$3 \cdot n$
3 additions	$3 \cdot n$
Bit shift (multiply by 2)	n
Test	n
Conditional jump	$3 \cdot n$
Total	$66 + 12 \cdot n$

Table 5.2. Detailed estimate of the execution time of the interpolation algorithm.

Operation	**Cycles**	**Time**
Vertical interpolation	503808	5 ms
Horizontal interpolation	8060928	81 ms
Frame buffer reads	2097152	21 ms
Bilinear filtering	19660800	197 ms
Frame buffer writes	524288	5 ms
Total	30846976	308 ms

Table 5.3. Optimistic estimate of the execution time of software texture mapping.

5.2.3 Total execution time

Using the above formula with $n = 15$, we can compute an estimate of the execution time of a software implementation (table 5.3).

1D interpolations need to be done twice, once for each texture coordinate.

The number of frame buffer reads is computed by considering that for each pixel written to the 512x512 destination picture, 4 pixels must be read from the source picture.

The cost of bilinear filtering is computed, for each destination pixel, with 4 subtractions, 8 multiplications, 4 additions and 1 bit shift times 3 color channels, which yields 75 cycles. This is optimistic as it does not take into account the time needed to decode the fixed point format.

According to this (yet optimistic) estimate, it becomes clear that a software implementation could not suffice, as the required performance is 8 ms. Even the vertical interpolation can hardly be implemented in software, as it would use alone more than 60% of the CPU power (which is needed for other tasks). We need an overall speedup by a factor of more than 40, using hardware acceleration.

5.3 Pipelined hardware implementation

5.3.1 Strategy

Given the performance constraints and the slowness of software implementations, we decided to implement the complete texture mapping process in hardware.

It is expected that the memory latency for reading the frame buffer would be a performance-limiting factor. Instead of trying to alleviate its effects by using complex and potentially resource-intensive techniques such as advanced prefetching or non-blocking caches, we simply use a direct-mapped blocking texel cache providing simplicity and fast hit times, and design the rest of the texture mapping unit so that the memory read latency becomes the *only* limiting factor.

With a direct-mapped texel cache having a hit rate of 90%, a hit time of 1 cycle and a miss penalty of 9 cycles, the average memory access times is 1.8 cycles. With a 100MHz system clock, such a cache has a throughput of 55 megapixels per second, well above the optimistic estimate made in subsection 5.2.1.[3]

To make sure that the memory access time is the only limiting factor, it was chosen that the rest of the system should be designed to support a throughput of approximately one output pixel per clock cycle. This heuristic was influenced by the fact that it corresponds to a spatial implementation of the algorithm (i.e. with little or no time-based resource sharing of the hardware components) but does not require resource-intensive duplication of large hardware units either. A spatial implementation requires more area than a time-shared one, but it is simpler to understand, and needs fewer multiplexers and is less prone to routing congestion, making it easier to achieve timing closure in FPGAs.

A deeply pipelined implementation of the texture mapping algorithm was thus chosen, whose block diagram is depicted in figure 5.5. Many of the stages have internal pipeline sub-stages, and they are detailed below.

5.3.2 Vertex fetch engine

There is not much to say about this stage, which is a straightforward finite state machine-based Wishbone bus master that fetches the texture coordinates of each vertex from the system memory, and sends them down the pipeline to the vertical interpolator.

The vertex fetch engine is connected to the lower-bandwidth Wishbone bus because this saves resources compared to FML (which has a wider data path) and makes it easier to handle cache coherency issues (section 4.5).

[3] This is a quick estimate assuming a normal cache that does not support bilinear filtering. To implement bilinear filtering, the situation is more complex as the cache needs to look up 4 pixels at once. This is discussed in subsection 5.3.6.

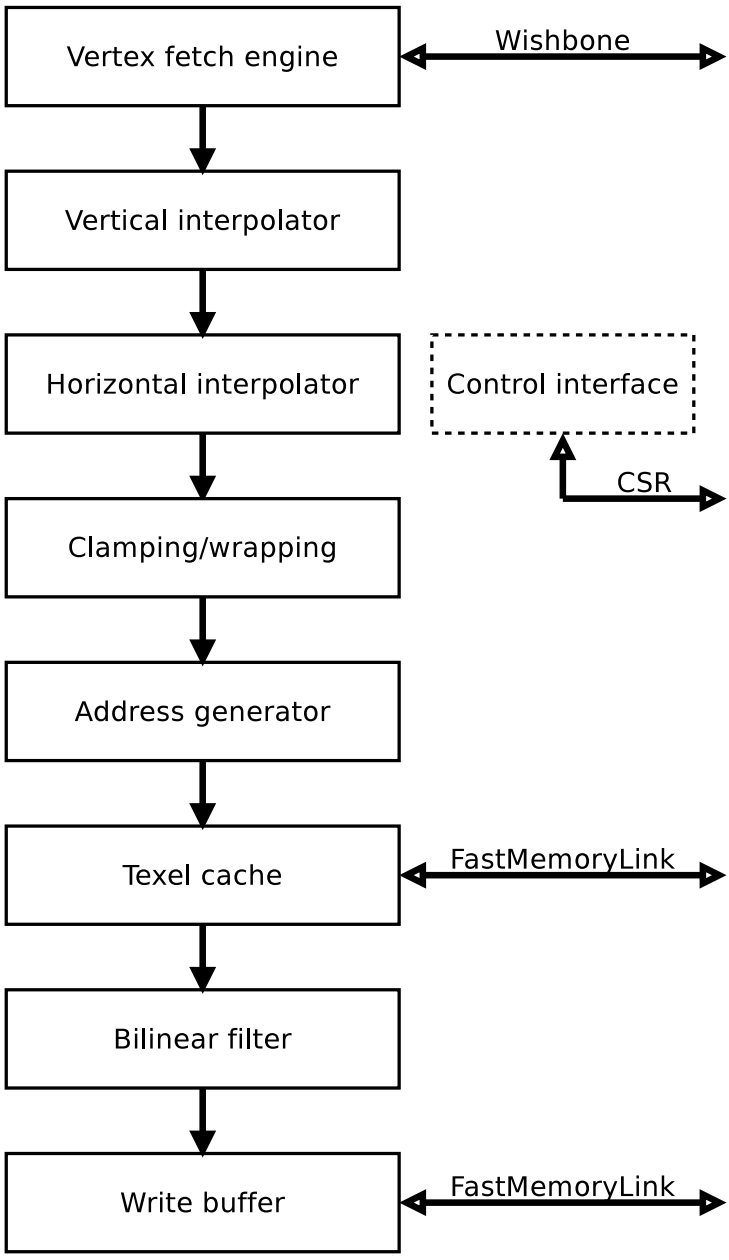

Figure 5.5. Block diagram of the texture mapping unit architecture.

5.3.3 Interpolators

The horizontal and vertical interpolators are both implemented in the same way. They are vector interpolators, which contain two scalar interpolators (figure 5.6, one for each texture coordinate). Each scalar interpolator contains additional internal pipeline sub-stages, as described in figure 5.7.

The vertical interpolator contains two vector interpolators, one for each vertical edge of the rectangle.

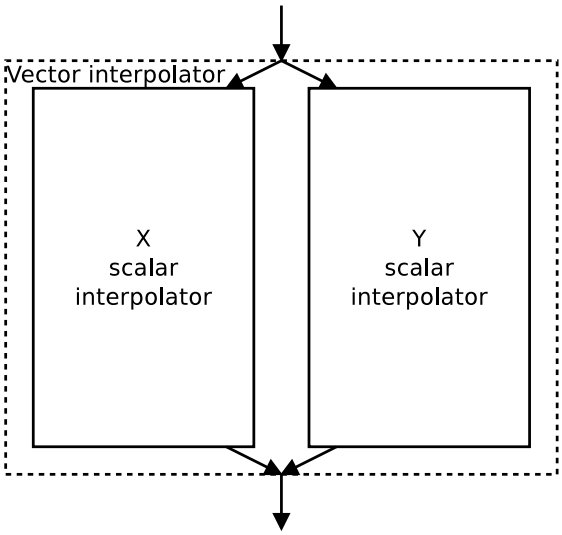

Figure 5.6. Vector interpolator.

Stages of the scalar interpolator

Stage A: Dx and Dy computation. This stage computes the two differences $y_1 - y_0$ and $x_1 - x_0$. It is based on simple registered arithmetic combinatorial functions, which compute the two differences in one clock cycle.

Stage B: Q and R computation. The next operation is to perform the Euclidean division of Dy by Dx. The hardware does so by using the restoring division method [19], which takes as many cycles as there are quotient digits (our implementation has 18).

In order to keep the resource usage low, the divider is not pipelined. After operands are sent to it, it stalls transmissions from the upstream stage for several cycles until the division is complete.

Stage C: Interpolation loop. Finally, the core of the algorithm (the "while" loop from figure 5.3) is implemented in the last stage. This unit receives the Q and R values from the dividers, as well as the start y_0 value and the range x_0 to x_1 (which

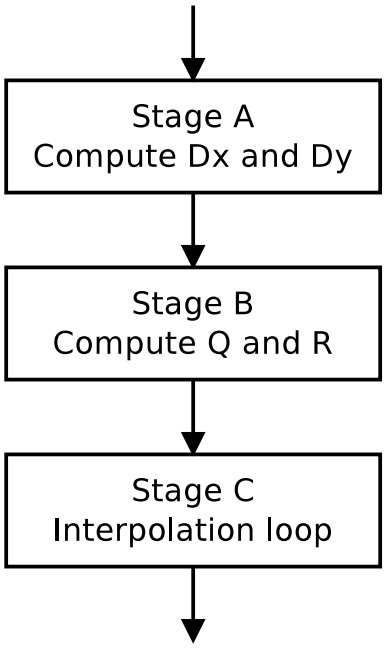

Figure 5.7. Pipelined scalar interpolator.

are forwarded through the previous stages). It then sends the series of interpolated values $[y]$ for $x_0 \leq x < x_1$.

The throughput of the stage is one interpolation point per clock cycle. While the interpolation is taking place, transmission of new parameters from the upstream stage is stalled. This justifies the choice of a slow but low-area restoring divider in stage B: with a typical rectangle size of 16, the processing times of the interpolation loop and of the divider are roughly the same, making the pipeline balanced.

5.3.4 Clamping/wrapping

This unit processes interpolated texture points whose coordinates are beyond the boundaries of the texture (i.e. they are negative or exceed the texture's horizontal or vertical resolution).

There are two, selectable, ways of dealing with them:

- *clamping*, which consists in replacing an out-of-range coordinate with 0 (if it was negative) or with the horizontal or vertical resolution of the texture minus one (if it was too large).

- *wrapping*, which repeats the texture and consists in computing the positive modulo of each coordinate with respect to the horizontal or vertical texture resolution. In order to avoid using an expensive fast divider, only textures whose sizes are a power of 2 are supported for wrapping. This enables the replacement of the divider with a bitwise AND operation, which is way less

expensive. The problem of negative texture coordinates is solved by simply masking out the sign bit, which yields the correct result as the coordinates are represented in two's complement format.

This stage is implemented by simple arithmetic combinatorial functions, which are registered and pipelined on two sub-stages to meet timing requirements.

5.3.5 Address generator

The address generator is a simple arithmetic circuit that turns the floating point texture coordinates into the four memory addresses of the pixels they cover, and the destination coordinate into the corresponding memory address in the destination frame buffer. It is pipelined on three sub-stages to meet timing goals.

The formula used to convert a coordinate (x, y) into a pixel address A within a 16bpp frame buffer starting at A_{base} and with an horizontal resolution H is the following:

$$A = A_{base} + (H \cdot y + x) \cdot 2 \tag{5.6}$$

5.3.6 Texel cache

Presentation

Once the addresses of the four texture pixels have been computed, the next step is to retrieve data from the memory. This should be done fast: to meet the performance goal of 31 megapixels per second at the output of the texture mapping unit, the texel cache must be able to fetch at least 124 megapixels per second. This is, on average, at least 1.24 pixel per clock cycle with a 100MHz system clock.

In consistence with the heuristic made at subsection 5.3.1 that consists in designing the system for a performance of one output pixel per clock cycle in the absence of memory read delays, the texel cache should be able to service the four request ports (called *channels*) in one clock cycle if all the channels hit the cache.

Channel are numbered as follows (see figure 2.8):

- Channel 1 fetches the *base* pixel, that is to say, the pixel at the coordinates obtained by flooring the non-integer texture coordinates. It is always active.
- Channel 2 fetches the pixel at the right of the base pixel. It is active when the X texture coordinate has a non-zero fractional part.
- Channel 3 fetches the pixel at the bottom of the base pixel. It is active when the Y texture coordinate has a non-zero fractional part.
- Channel 4 fetches the pixel at the bottom-right of the base pixel. It is active when both the X and Y texture coordinate have a non-zero fractional part.

Separate vs. shared caches

The obvious solution seems have one separate cache per channel. However, this solution is not optimal in terms of speed and memory efficiency. For example, let us take the case when the texture mapping consists in zooming the texture by a factor of 2 (the texture coordinates at each vertex are the vertex coordinates divided by 2). Assuming an empty cache at the beginning, the sequence of events is as follows:

1. The interpolated fixed-point texture coordinates are (0, 0). Channel 1 misses its cache for a fetch of the pixel at (0, 0). Since the coordinates are integer, channels 2, 3 and 4 are idle and do not need to fetch data.

2. The texture coordinates become (0.5, 0). Channel 1 hits its cache for the pixel at (0, 0). However, channel 2 misses its cache for the pixel at (1, 0) and a new memory request needs to be performed, even though the pixel at (1, 0) is in the cache of channel 1 (it was part of the burst that fetched the (0, 0) pixel). Channels 3 and 4 are idle, since the Y coordinate is integer.

The problem repeats every time the X texture coordinate crosses a memory burst boundary, and is also present in the Y direction with channels 3 and 4. In total, the texel cache uses *four times* as much memory bandwidth as it would use if it were able to share data between the channels' respective caches.[4] Zooming (locally or globally) is a very common operation, so the issue needs to be addressed.

A more efficient solution, which has been retained, consists therefore in having a single multi-ported data store.

Implementation

Our implementation is based on the traditional direct-mapped cache, but using quad-port SRAM for the data and tag stores. Quad-port SRAM can be mapped to FPGA technologies at a moderate cost by using two primitive dual-port SRAMs in which the data is replicated. During normal operation (hits), each port serves one channel, and, when refilling the cache on a miss, reading is disabled and two of the ports (one per primitive dual-port SRAM) are used to feed the data into the RAMs.

A simplified block diagram of the texel cache is given in figure 5.8. This block diagram does not include all of the logic needed to handle pipeline stalls and lacks the "valid" bits of the tags.

At each clock cycle, the texel cache processes, in a pipelined manner,[5] four memory addresses from each channel if they hit the caches. The "hit" signal is kept high and the pipeline is always running.

In case of a miss, the "hit" signal goes low (stalling the pipeline), and the priority encoder and the multiplexer (MUX) select one of the missed addresses (there can

[4] Assuming at least two complete horizontal lines of pixels from a primitive rectangle fit in the cache, which is generally the case.

[5] The SRAMs are registered, in order to improve timing and to map to the block RAMs of common modern FPGAs which always contain an internal register.

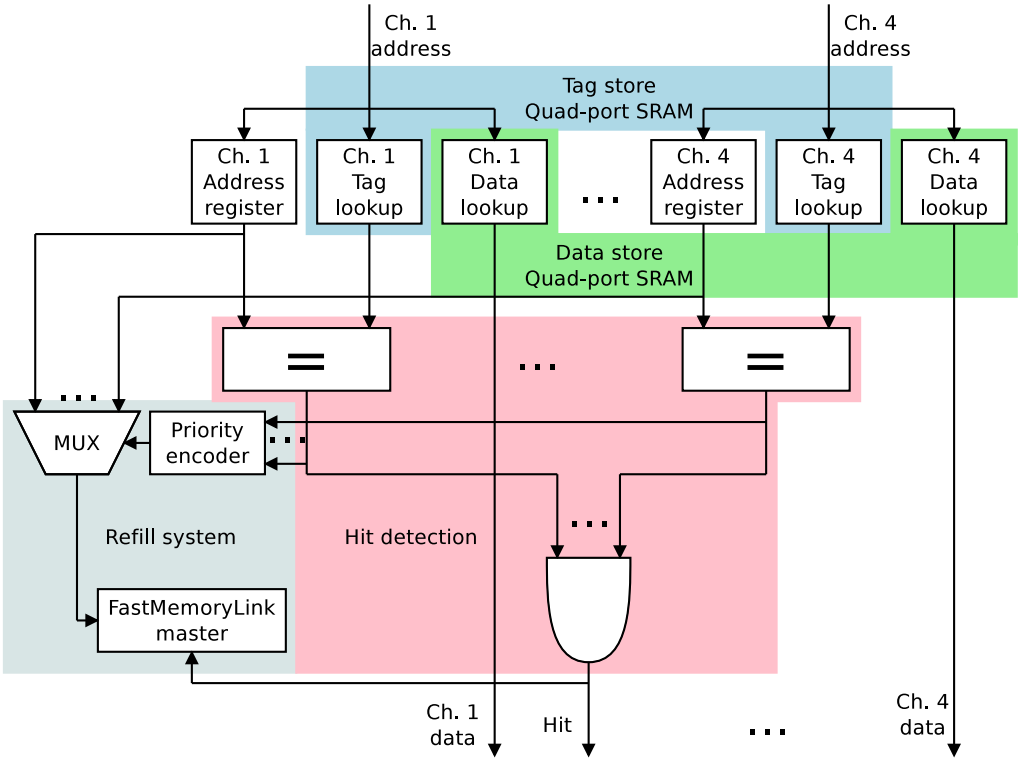

Figure 5.8. Architecture of the four-channel texel cache.

be one or many). The FastMemoryLink master issues a memory transaction to retrieve the data from the system memory, replaces the contents of the cache line and rewrites the tag. The address now becomes a cache hit. If no other address misses the cache, the texel cache has successfully handled the 4-channel transaction and the "hit" signal goes high again to proceed to the next. Otherwise, the process repeats until all addresses hit the cache. Our design does not take advantage of the pipelining feature of the FastMemoryLink bus and issues requests sequentially.

Inter-channel cache conflicts

An *inter-channel cache conflict* (ICCC) occurs when two or more channels request different addresses that have different tags but map to the same cache line.

This condition is not desirable. With our implementation, the texel cache would go into an infinite loop fetching data from the memory in an attempt to make all channels hit the cache — which it can never achieve — until it is manually reset.[6]

This choice has been made for two reasons: first, adding hardware to deal with ICCCs would yield poor performance anyway as some memory bursts would be there only for retrieving one pixel and solving the conflict,[7] second, ICCCs are easy to avoid for our purposes, and we will see how.

For simplicity, we use the pixel (2 bytes) as unit. In the equations that follow:

- H is the horizontal texture resolution in pixels.
- V is the vertical texture resolution in pixels.
- N_l is the number of pixels a cache line can hold. It is equal to the line size in bytes divided by 2.
- N_c is the total number of pixels the texel cache can hold. It is equal to the cache size in bytes (not counting the tag memory) divided by 2.

Characterization of cache conflicts. A pixel at address a_p (measured in pixels, i.e. 2-byte words) is mapped to the cache line indexed by:

$$l_p = \left\lfloor \frac{a_p}{N_l} \right\rfloor \pmod{\frac{N_c}{N_l}} \tag{5.7}$$

[6] As a safety measure, it is therefore recommended that software drivers for the texture mapping unit check for the possibility of ICCC conditions before running the TMU and report an error if an ICCC is possible.

[7] This is true only if we keep a direct-mapped cache. With a multiple-way set-associative cache and a smart replacement policy that allocates one specific way to each conflicting channel when an ICCC occurs, the hardware can both deal with ICCCs and yield high performance. However, it is more complex and expensive. Furthermore, when keeping a direct-mapped cache, it makes sense to add hardware that would deal with infrequent cases of ICCCs such as those arising when wrapping at texture boundaries.

Thus, two pixels at addresses a_1 and a_2 conflict in the cache if and only if:

$$\begin{cases} |a_1 - a_2| \geq N_l \\ \left\lfloor \frac{a_1}{N_l} \right\rfloor \equiv \left\lfloor \frac{a_2}{N_l} \right\rfloor \pmod{\frac{N_c}{N_l}} \end{cases} \tag{5.8}$$

Texture clamping only causes one or more channel addresses to be equal, and therefore does not introduce additional cases of ICCCs. However, texture wrapping does introduce new dispositions of the channels within the texture, and new ICCC conditions.

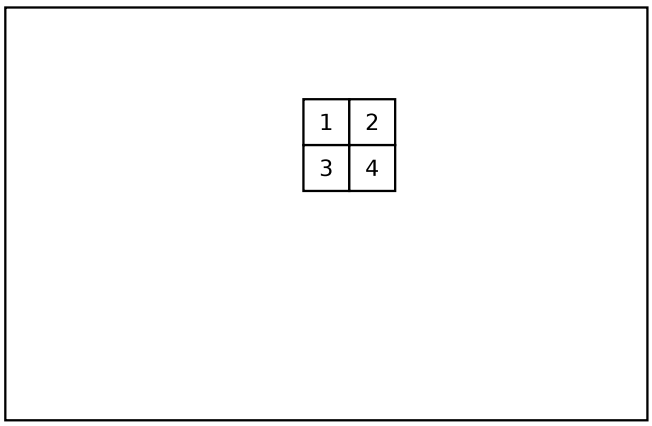

Figure 5.9. Disposition of the channels within the texture, general case.

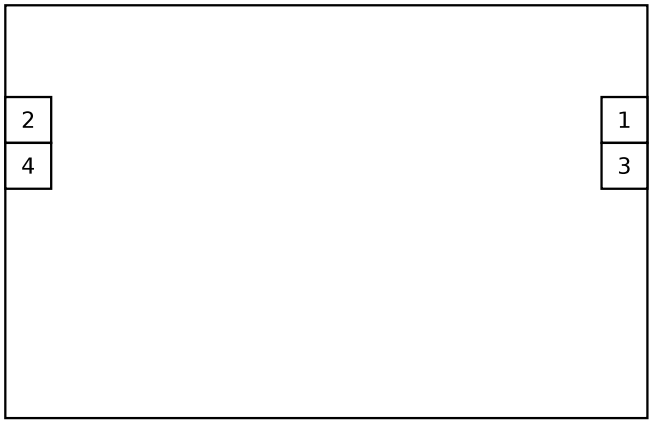

Figure 5.10. Disposition of the channels within the texture, vertical wrapping.

Conflicts between channels 1 and 2 (or 3 and 4). The addresses a_A and a_B of these two channels can be separated by either:

- 1 pixel in the most common case (sampling in the middle of the texture, see figure 5.9). This cannot cause inter-channel cache conflicts.

- $H-1$ pixels if texture wrapping is enabled and the texture is sampled at one of its vertical edges (figure 5.10). In this case, the condition $|a_A - a_B| \geq N_l$ is often verified (except for small textures where $H-1 < N_l$). To make sure that there will be no ICCC, we must thus make sure that the following condition is also verified:

$$\left\lfloor \frac{a_A}{N_l} \right\rfloor \not\equiv \left\lfloor \frac{a_A + H - 1}{N_l} \right\rfloor \pmod{\frac{N_c}{N_l}} \tag{5.9}$$

To make sure that this condition is verified for all possible pixel addresses, it is sufficient to check that:

$$\forall a \in \{0, 1, ...N_l - 1\}, \left\lfloor \frac{a + H - 1}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \tag{5.10}$$

Indeed, by division by N_l we have $a_A = k \cdot N_l + a$ with $0 \leq a \leq N_l - 1$, which transforms equation 5.9 into:

$$k + \left\lfloor \frac{a}{N_l} \right\rfloor \not\equiv k + \left\lfloor \frac{a + H - 1}{N_l} \right\rfloor \pmod{\frac{N_c}{N_l}} \tag{5.11}$$

which leads easily to the result, considering that $\left\lfloor \frac{a}{N_l} \right\rfloor = 0$.

This can be further simplified:

$$\begin{cases} \left\lfloor \frac{H-1}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \\ 1 + \left\lfloor \frac{H-1}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \end{cases} \tag{5.12}$$

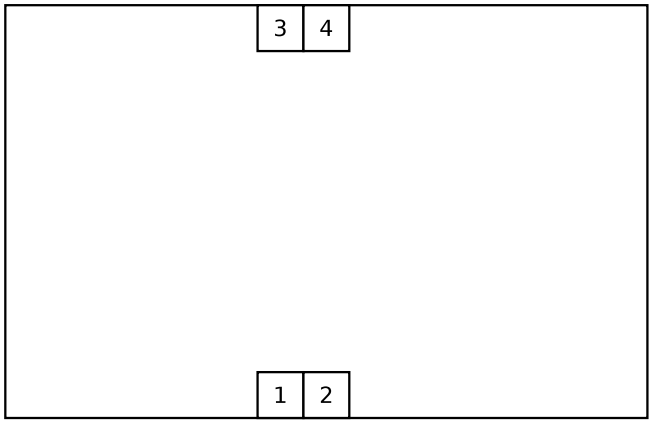

Figure 5.11. Disposition of the channels within the texture, horizontal wrapping.

Conflicts between channels 1 and 3 (or 2 and 4). The separation between the channels' addresses can be:

- H pixels (according to the equation 5.6) in the general case (figure 5.9). Using the same reasoning from above, we can deduce that it is sufficient to check that $H < N_l$ or:

$$\begin{cases} \left\lfloor \frac{H}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \\ 1 + \left\lfloor \frac{H}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \end{cases} \tag{5.13}$$

- $H \cdot (V-1)$ pixels if texture wrapping is enabled and the texture is sampled at one of its horizontal edges (figure 5.11). Again, it is sufficient to check that $H \cdot (V-1) < N_l$ or:

$$\begin{cases} \left\lfloor \frac{H \cdot (V-1)}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \\ 1 + \left\lfloor \frac{H \cdot (V-1)}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \end{cases} \tag{5.14}$$

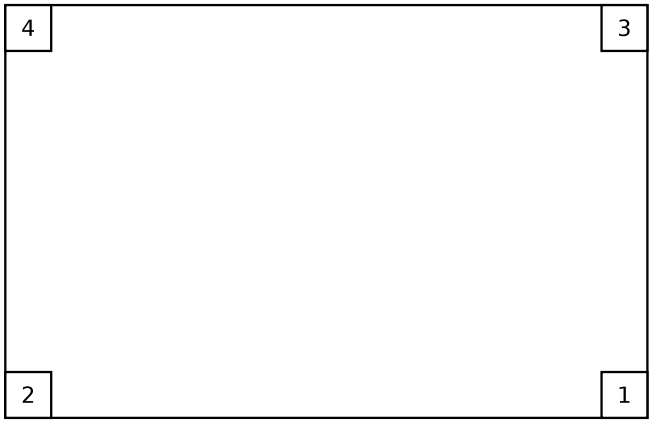

Figure 5.12. Disposition of the channels within the texture, horizontal and vertical wrapping.

Conflicts between channels 1 and 4. The channels' addresses can be separated by:

- $H + 1$ pixels (according to the equation 5.6) in the general case (figure 5.9). It is therefore sufficient to check that $H + 1 < N_l$ or:

$$\begin{cases} \left\lfloor \frac{H+1}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \\ 1 + \left\lfloor \frac{H+1}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \end{cases} \tag{5.15}$$

- 1 pixel in case of vertical wrapping (figure 5.10). This cannot cause ICCCs.

- $H \cdot (V-1) - 1$ pixels in case of horizontal wrapping (figure 5.11). Similarly, we can check that $H \cdot (V-1) - 1 < N_l$ or:

$$\begin{cases} \left\lfloor \frac{H \cdot (V-1) - 1}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \\ 1 + \left\lfloor \frac{H \cdot (V-1) - 1}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \end{cases} \tag{5.16}$$

- $H \cdot V - 1$ in case of wrapping in both directions (figure 5.12). We can check that $H \cdot V - 1 < N_l$ or:

$$\begin{cases} \left\lfloor \frac{H \cdot V - 1}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \\ 1 + \left\lfloor \frac{H \cdot V - 1}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \end{cases} \tag{5.17}$$

Conflicts between channels 2 and 3. Like above, we can check that $H - 1 < N_l$ or:

$$\begin{cases} \left\lfloor \frac{H - 1}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \\ 1 + \left\lfloor \frac{H - 1}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \end{cases} \tag{5.18}$$

Furthermore, if texture wrapping is enabled, the differences between pixel addresses can become (resulting in similar conditions to check for):

- $2 \cdot H - 1$ (vertical wrapping, figure 5.10).
- $H \cdot (V - 1) + 1$ (horizontal wrapping, figure 5.11).
- $H \cdot (V - 2) + 1$ (wrapping in both directions, figure 5.12).

Summary. To avoid inter-channel cache conflicts, when texture clamping is used, it is sufficient to check that (all equivalences are modulo $\frac{N_c}{N_l}$):

$$\boxed{\begin{cases} H - 1 < N_l & \text{or} \begin{cases} \left\lfloor \frac{H-1}{N_l} \right\rfloor \not\equiv 0 \\ 1 + \left\lfloor \frac{H-1}{N_l} \right\rfloor \not\equiv 0 \end{cases} \\ H < N_l & \text{or} \begin{cases} \left\lfloor \frac{H}{N_l} \right\rfloor \not\equiv 0 \\ 1 + \left\lfloor \frac{H}{N_l} \right\rfloor \not\equiv 0 \end{cases} \\ H + 1 < N_l & \text{or} \begin{cases} \left\lfloor \frac{H+1}{N_l} \right\rfloor \not\equiv 0 \\ 1 + \left\lfloor \frac{H+1}{N_l} \right\rfloor \not\equiv 0 \end{cases} \end{cases}} \tag{5.19}$$

If texture wrapping is used, additional conditions appear:

$$\begin{cases} H \cdot (V-1) - 1 < N_l & \text{or} \begin{cases} \left\lfloor \frac{H \cdot (V-1)-1}{N_l} \right\rfloor \not\equiv 0 \\ 1 + \left\lfloor \frac{H \cdot (V-1)-1}{N_l} \right\rfloor \not\equiv 0 \end{cases} \\ H \cdot (V-1) < N_l & \text{or} \begin{cases} \left\lfloor \frac{H \cdot (V-1)}{N_l} \right\rfloor \not\equiv 0 \\ 1 + \left\lfloor \frac{H \cdot (V-1)}{N_l} \right\rfloor \not\equiv 0 \end{cases} \\ H \cdot (V-1) + 1 < N_l & \text{or} \begin{cases} \left\lfloor \frac{H \cdot (V-1)+1}{N_l} \right\rfloor \not\equiv 0 \\ 1 + \left\lfloor \frac{H \cdot (V-1)+1}{N_l} \right\rfloor \not\equiv 0 \end{cases} \\ H \cdot (V-2) + 1 < N_l & \text{or} \begin{cases} \left\lfloor \frac{H \cdot (V-2)+1}{N_l} \right\rfloor \not\equiv 0 \\ 1 + \left\lfloor \frac{H \cdot (V-2)+1}{N_l} \right\rfloor \not\equiv 0 \end{cases} \\ 2 \cdot H - 1 < N_l & \text{or} \begin{cases} \left\lfloor \frac{2 \cdot H-1}{N_l} \right\rfloor \not\equiv 0 \\ 1 + \left\lfloor \frac{2 \cdot H-1}{N_l} \right\rfloor \not\equiv 0 \end{cases} \\ H \cdot V - 1 < N_l & \text{or} \begin{cases} \left\lfloor \frac{H \cdot V-1}{N_l} \right\rfloor \not\equiv 0 \\ 1 + \left\lfloor \frac{H \cdot V-1}{N_l} \right\rfloor \not\equiv 0 \end{cases} \end{cases} \tag{5.20}$$

Cache size

We must now determine an optimal size for the texel cache. The size must represent a compromise between hit rate (and, thus, performance) and costly utilization of on-chip RAM. Furthermore, it must be chosen so that inter-channel cache conflicts do not occur for our use cases.

To study the impact of the cache size on the hit rate, we simulated the complete Verilog code of texture mapping unit with different cache sizes. The CSR interface of the texture mapping unit (subsection 5.3.9) supports eight registers that measure, for each channel, the number of requests[8] and how many of those requests hit the cache. Those registers are still present after logic synthesis, and can be used to validate the performance of the texel cache in the real system.[9]

The source and destination images have a resolution of 512x512 pixels, and are tessellated with 16x16 rectangles forming a 32x32 mesh — which matches the configuration used for distortions by the renderer program. Different sets of texture coordinates were used at each vertex, according to table 5.4. Texture coordinates are multiplied by 64 to perform the conversion of integers into the fixed point format

[8] As outlined above, channels are only active (i.e. issue a cache request) when needed (i.e. when the interpolated texture coordinates are not integer). Channel 1 is always active and therefore makes as many accesses as there are pixels in the output picture. Its accesses are however still counted as a means to check that the texture mapping unit is operating correctly.

[9] The reason why we performed the tests using Verilog simulations instead of the real FPGA-based system is because logic synthesis — needed for each tested cache size — takes a long time, and some cache configurations may cause resource shortages, timing problems or even Xst synthesizer bugs that would need to be manually addressed.

Set name	**Output picture**	**X**	**Y**
copy	Figure 5.13	$x \cdot 16 \cdot 64$	$y \cdot 16 \cdot 64$
zoomin	Figure 5.14	$x \cdot 10 \cdot 64$	$y \cdot 10 \cdot 64$
zoomout	Figure 5.15	$x \cdot 40 \cdot 64$	$y \cdot 40 \cdot 64$
rotozoom	Figure 5.16	$(x \cdot 16 - 256) \cdot 67 - (y \cdot 16 - 256) \cdot 28 + 256 \cdot 64$	$(x \cdot 16 - 256) \cdot 28 + (y \cdot 16 - 256) \cdot 67 + 256 \cdot 64$

Table 5.4. Texture coordinate sets used for benchmarking the texel cache.

used throughout the linear interpolation process (see subsection 5.1.3). x and y refer to the indices of the vertex in the mesh. Texture wrapping was enabled, as it puts more load on the cache than texture clamping.

The sets were chosen for the following reasons:

- *copy* does not introduce any distortion and is meant to test the performance of the TMU as a blitter, that could be used e.g. for GUI acceleration. It is also a very simple case that verifies the consistency of the results.
- *zoomin* scales up the picture. This tests the TMU in a favorable case: as the amount of zoom is important, the texels are swept across slower than output pixels are drawn. This is expected to generate a lot of cache hits.
- *zoomout* scales down and repeats the picture. This is a detrimental case: the texels are swept across faster than the output pixel are drawn, and some values from the cache lines read from memory will have to be discarded. This is expected to generate a lot of cache misses.
- *rotozoom* slightly scales down and rotates the picture. This is an intermediate case that reflects what MilkDrop typically does: rotation and slight scale-down are common preset effects.

The length of the cache line is set to the length of a FML burst in the SoC (4 words of 64 bits each) for simplicity of the design.

Simulations were carried out using the free GPL Cver Verilog simulator [31]. To visually inspect the results of the distortions stemming from each set of texture coordinates, the simulation reads and writes input and output picture files thanks to a Verilog VPI [10] plug-in.[10] A typical simulation trace is reproduced in figure 5.17. Even though GPL Cver is relatively slow[11] to carry out such a complex simulation, it produced consistent results, which supports the idea that it is, in many cases, a viable alternative to proprietary and expensive simulators commonly taught in university courses.

Results are reported in table 5.5 and figure 5.18. Only the *global* hit rate is reported, which is computed as the global number of hits over the global number of

[10] This was also part of the usual test bench used to debug the Verilog implementation of the texture mapping unit.

[11] Its runtime is between one and three minutes on a Intel Core 2 Duo 2.5GHz machine.

Figure 5.13. TMU output picture for the "copy" set (original picture).

Figure 5.14. TMU output picture for the "zoomin" set.

Figure 5.15. TMU output picture for the "zoomout" set.

Figure 5.16. TMU output picture for the "rotozoom" set.

```
$ make TB=tb_tmu2.v
cver +loadvpi=./vpi_images.so:vpi_register tb_tmu2.v
(...)
GPLCVER_2.12a of 05/16/07 (Linux-elf).
Copyright (c) 1991-2007 Pragmatic C Software Corp.
  All Rights reserved.  Licensed under the GNU General Public
  License (GPL).
  See the 'COPYING' file for details.  NO WARRANTY provided.
Today is Tue May  4 14:46:22 2010.
PLI Image I/O functions registered
Compiling source file "tb_tmu2.v"
Compiling source file "../rtl/tmu2_adrgen.v"
(...)

Opening input picture...
Configuring TMU...
CSR write: 0000002c=01000000
CSR write: 00000004=00000020
(...)
CSR write: 00000044=00000010
CSR write: 00000048=0000003f
Starting TMU...
CSR write: 00000000=00000001
Received DONE IRQ from TMU!
Gathering texel cache statistics...
CSR read : 00000050=00040000
CSR read : 00000054=0003c000
(...)
CSR read : 00000068=00000000
CSR read : 0000006c=00000000
Channel A:      245760 /      262144 hits (93.750000 %)
Channel B:           0 /           0 hits (nan %)
Channel C:           0 /           0 hits (nan %)
Channel D:           0 /           0 hits (nan %)
GLOBAL   :      245760 /      262144 hits (93.750000 %)
Writing output picture...
All done!
Halted at location **tb_tmu2.v(367) time 4585546 from call
to $finish.
  There were 0 error(s), 32 warning(s), and 402 inform(s).
```

Figure 5.17. Typical TMU simulation trace (excerpt).

Cache size	copy	zoomin	zoomout	rotozoom
2kB	93.75 %	98.08 %	83.33 %	73.06 %
4kB	93.75 %	98.08 %	83.33 %	82.94 %
8kB	93.75 %	98.62 %	83.33 %	93.73 %
16kB	93.75 %	99.27 %	86.94 %	95.74 %
32kB	93.75 %	99.27 %	91.44 %	96.02 %
64kB	93.75 %	99.27 %	94.14 %	96.02 %
128kB	93.75 %	99.27 %	94.14 %	96.15 %

Table 5.5. Hit rates for each set of texture coordinates and different cache sizes.

accesses. The global number of hits (respectively accesses) is the sum of the number of hits (respectively accesses) for all channels. The reported cache size is the size of the data store only (not counting the tag memory).

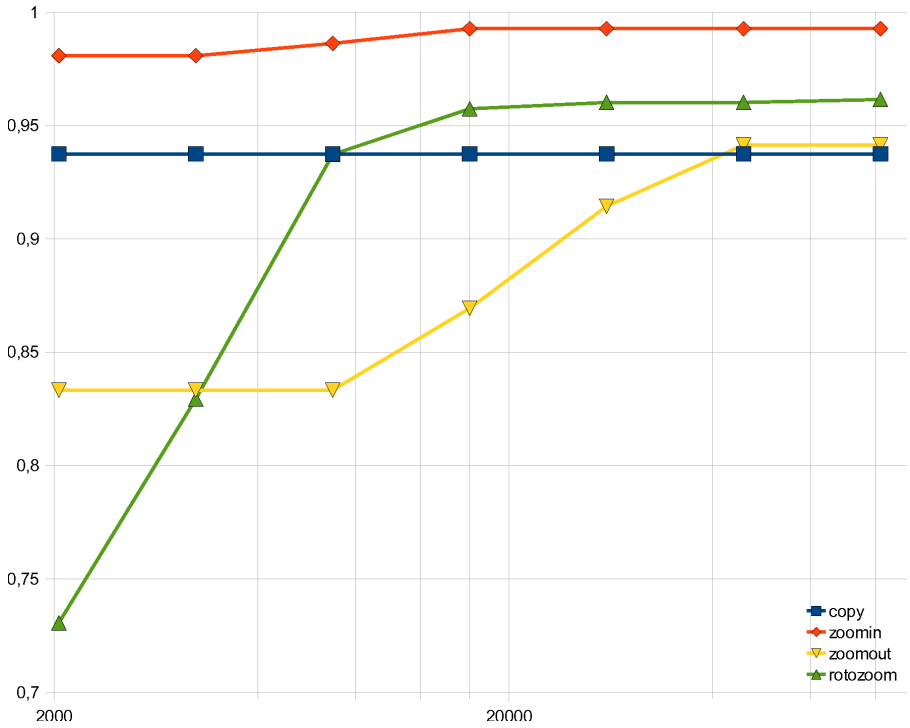

Figure 5.18. Hit rates versus texel cache size. The X axis (cache size) uses a logarithmic scale.

Before we go on choosing a texel cache size, a few comments on these results can be made.

First, for the "copy" set, the hit rate remains at a constant 93.75 %. This is expected and supports the validity of the results. Indeed, the texture mapping unit is copying rectangles whose horizontal size is the length in pixels of a cache line (4 words of 64-bit each yields 16 pixels of 16 bits each). In our tests, the

texture frame buffer was aligned to the beginning of a cache line and 512 (the horizontal texture size) is a multiple of the cache line length in pixels, therefore each horizontal line in a rectangle corresponds to a cache line. Each texture pixel is read exactly once (channels 2, 3 and 4 are idle as the interpolated texture coordinates are always integer). What happens is that, for each horizontal line in each rectangle, the first texel is read and misses the cache. Then, the next 15 texels in the line are immediately read and hit the cache. This indeed yields a cache hit rate of $\frac{15}{16} = 0.9375$.

A result more surprising at first sight is that for the "zoomout" set, which is supposed to be the worst case, the cache hit rate is low — as expected — and increases, until it slightly exceeds the hit rate for the "copy" set for cache sizes of 64kB and above.

This has to do with the fact that the texture mapping unit draws the rectangles in the same order as frame buffers are scanned (from left to right and top to bottom). When the cache reaches 64kB, it is able to memorize a full band of texels that is more than 48 pixels high ($512 \cdot 48 \cdot 2 = 49152 < 65536$) which contains more than all the texels needed to draw a full line of rectangles (16 pixels high) in the output picture. Since the output picture repeats the texture, the repetition generates cache hits that are not present with the "copy" set.

As for choosing a texel cache size, we went for a 32kB cache. There is no significant performance improvement with using a larger cache except for the "zoomout" set for which there is a slight increase in the hit rate between 32kB and 64kB. Since the "zoomout" set is a worst-case scenario seldom found in practice (MilkDrop usually zooms out by factors much less than the one we used, resulting in fewer cache misses), we felt it was not worth doubling the cache size to improve its performance.

We also need to check that this cache size cannot cause inter-channel cache conflicts (ICCCs). The cache line size is 32 bytes, thus, the cache line length in 16bpp pixels is $N_l = 16$ and the cache holds $\frac{N_c}{N_l} = 1024$ lines. The texture mapping unit is operated with 512x512 textures and texture wrapping is enabled.

With these parameters, all the conditions of equations 5.19 and 5.20 are verified, except one: we have $1 + \left\lfloor \frac{H \cdot V - 1}{N_l} \right\rfloor \equiv 0$, which means that there can be inter-channel cache conflicts between channels 1 and 4 due to wrapping. Without additional hypotheses, this can only be solved by increasing the cache size to at least 1MB, which would be very expensive.

Fortunately, there is a cheaper solution. What we actually need to verify is this condition, obtained in the same way as equation 5.9:

$$\left\lfloor \frac{a_A}{N_l} \right\rfloor \not\equiv \left\lfloor \frac{a_A + H \cdot V - 1}{N_l} \right\rfloor \pmod{\frac{N_c}{N_l}} \tag{5.21}$$

We *add* the hypothesis that the frame buffer address is a multiple of the cache line length, which we can easily achieve by imposing alignment requirements to the software tool chain. Without loss of generality, we suppose that a_A is the address of channel 4, which is equal to the address of the frame buffer in the case creating the

problem (figure 5.12). $\frac{a_A}{N_l}$ is therefore integer, and the only condition that implies the absence of conflict between channels 1 and 4 becomes:

$$\left\lfloor \frac{H \cdot V - 1}{N_l} \right\rfloor \not\equiv 0 \pmod{\frac{N_c}{N_l}} \tag{5.22}$$

This condition is verified. So, ICCCs will always be avoided with a 32kB texel cache if we make sure that the 512x512 texture is aligned to a 32-byte boundary.

5.3.7 Bilinear filter

The bilinear filter is a straightforward arithmetic circuit that implements the equation 5.5 with five pipeline sub-stages.

5.3.8 Write buffer

The write buffer is in charge of gathering the final pixels produced by the algorithm, assemble them into FML bursts and write them to the memory.

The write buffer has enough memory capacity to store two bursts on-chip. This storage space is used for a "double buffering" technique: while the first burst buffer receives pixels from the pipeline, the other buffer can transmit data to the memory controller. Bursts can be complete, which means that all data in them is valid, or incomplete, which means that the write buffer has not received a value for all the pixels within the burst but still needs to move the partial burst data it has off-chip because it does not have space to store it. Incomplete bursts are performed using the data mask (DM) signals of FML that prevent some bytes from being written to the memory during the burst. Incomplete bursts should be avoided as they waste memory bandwidth and reduce performance.

This small amount of on-chip memory is enough to perform well. Indeed, the algorithm scans the rectangles one by one horizontally and then vertically (figure 5.2) and consecutive pixels on the same horizontal line are contiguous in memory (equation 5.6). Thus, if the destination frame buffer is aligned to the start of a FML burst and if the horizontal size of the rectangles is a multiple of the number of pixels in a FML burst, the burst buffers will be used very efficiently, with complete bursts only. This is the recommended mode of operation for the texture mapping unit.

Under this assumption, what limits the throughput of the write buffer is the time it takes to empty the second burst buffer into the memory. This time is equal to the memory write access time plus the length of the FML burst.

In the equations that follow, these symbols are used:

- f is the system clock frequency in Hz.
- w is the width of a FML word in bits.
- n is the FML burst length.
- Δ_w is the memory write access time.

- d is the number of bits per pixel.
- T is the throughput of the write buffer, in pixels per second.

We therefore have:

$$T = \frac{f \cdot n \cdot w}{d \cdot (\Delta_w + n)} \tag{5.23}$$

Thus, the write buffer can achieve a throughput of one pixel per clock cycle ($T = f$) if the memory write access time verifies:

$$\Delta_w \leq \frac{n \cdot w}{d} - n \tag{5.24}$$

In Milkymist, the color format uses 16 bits per pixel and the FML bus is based on bursts of four 64-bit words, which leads to the throughput plot of figure 5.19.[12] The write buffer can tolerate write latencies of up to 12 cycles while maintaining an excellent performance of one pixel per clock cycle, which seems achievable even when taking into account the delays due to bus arbitration. Beyond this point, performance drops.

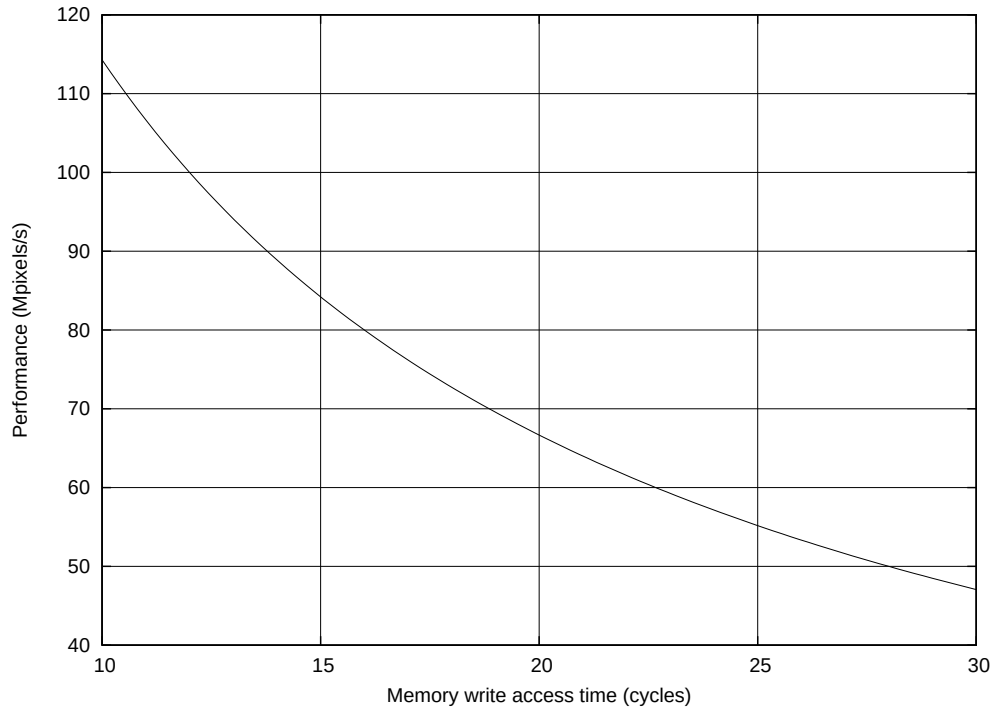

Figure 5.19. Theoretical write buffer throughput versus memory write access time.

[12] In our implementation, the throughput is also limited to a maximum of 100 megapixels per second because of the width of the write buffer input port.

5.3.9 Control interface

The texture mapping unit is completely under software control thanks to a CSR interface through which the CPU can configure and control it using a set of configuration and status registers. The texture mapping unit has one interrupt line to signal completion of the process to the CPU.

5.4 Extra features

Beyond this basic principle of operation, the texture mapping unit supports several features which are implemented as additional pipeline stages (not shown in figure 5.5):

- Fade to black. To implement the "decay" effect of MilkDrop, the output picture can be darkened by multiplying all its color components by a 6-bit fixed point number between 0 and 1.
- Chroma key. Texels of a given color can be ignored (not drawn in the output frame buffer). This is not used in normal rendering, but makes it possible to draw quickly text or symbols with transparent areas on the screen.
- Semi-transparency (alpha blending). The output can be made semi-transparent with 64 transparency levels. This is accomplished by reading the destination picture, mixing each pixel with the output of the texture mapping (by computing a weighted average) and writing the result back to memory. If transparency is not desired, the texture mapping unit skips reading the destination picture in order to use less memory bandwidth and avoid blocking on unneeded memory references.

5.5 Implementation results

We were unable to implement the planned 32kB texel cache in the XC4VLX25 FPGA of our ML401 development board, because the complete SoC with such a cache exceeds the on-chip SRAM capacity of the FPGA. Therefore, for these experiments, we had to use a 16kB cache instead. This issue should be resolved easily in the future, as our final board (chapter 8) will have an FPGA with much more on-chip memory.

We wanted to validate the performance of our texture mapping unit (TMU) design, measured in megapixels per second at the output (also called *fill rate*). Since it is a memory-bound process (subsection 5.3.1), it is relevant to examine its performance for different values of the texel cache hit rate.

To do so, we varied the texel cache hit rate by making the TMU zoom a picture at different levels. We proceeded with a texture size of 512x512 and an output picture of 640x480. The vertex mesh had a resolution of 32x32, and, to implement the different zoom levels, the vertex coordinates were $(z \cdot x, z \cdot y)$ with z varying between

0 and 2047. Each measurement was done twice to reduce the risk of errors and transients (CPU interrupts, DRAM refreshes, etc.), yielding 4096 points which are plotted in figure 5.20. The measurements were done programmatically by a software routine running on the system-on-chip itself. The video output was enabled during the process (and showing the result of the texture mapping), running in the standard VGA mode of 640x480 at 60Hz and putting a background load of approximately 300 Mb/s on the memory system for scanning the frame buffer.

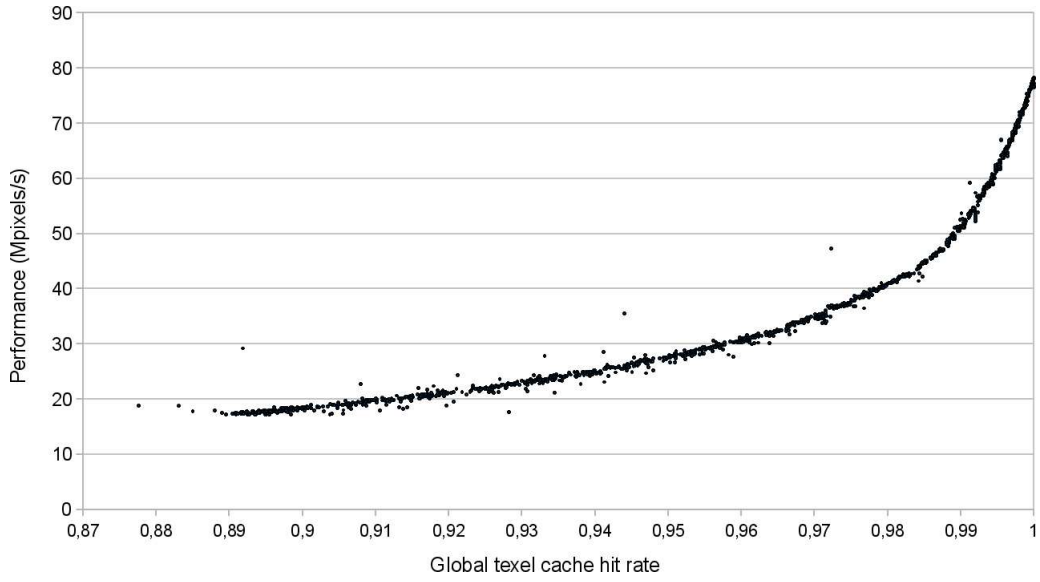

Figure 5.20. Measured TMU performance versus global texel cache hit rate.

The plot underlines the importance of the memory subsystem in a performance-driven texture mapping unit design. Indeed, performance drops sharply as soon as many off-chip memory references begin to be made (especially between hit rates of 100% and 98.5%). Our texture mapping unit is unable to meet its initial performance goal of 31 megapixels per second for hit rates below approximately 96%. Prefetching would be a good way to improve this result [16], but it comes at the cost of an increased hardware complexity.

However, our design still appears suitable for the application of rendering MilkDrop-like patches with all the options enabled (i.e. in the worst case) at VGA resolution (640x480). Indeed, this would involve:

1. Distortion of a 512x512 texture to a 512x512 texture. This is represented by the "rotozoom" set of vertex coordinates (table 5.4) whose resulting cache hit rate is 95.74 % (table 5.5). The fill rate is therefore approximately 30 megapixels per second (figure 5.20). The time taken by this process is thus $\frac{512 \cdot 512}{30 \cdot 10^6} = 8.7\text{ms}$.

2. Inclusion of a live video frame into the texture.[13] We assume the input video frame to be 720x288 (we are using *bob de-interlacing*, i.e. line doubling of the interlaced video frames) and the target rectangle in the texture to be 360x288. This is represented by the "zoomout" set with a cache hit rate of 86.94 % yielding a fill rate of very approximately 15 megapixels per second. The time taken by this process is $\frac{360 \cdot 288}{15 \cdot 10^6} = 6.9\text{ms}$.

3. Zooming of the 512x512 texture to 640x480 resolution, done twice (once for the normal picture and once for video echo[14]). This is represented by the "zoomin" set of vertex coordinates. The texel hit rate is estimated at 99.27%, and the fill rate at 55 megapixels per second. The time taken is estimated to be $2 \cdot \frac{640 \cdot 480}{55 \cdot 10^6} = 11.2\text{ms}$.

The total time is 26.8ms, which corresponds to 37 frames per second. This is more than enough to achieve a smooth video animation.

[13] We are — very optimistically — neglecting the time it takes to read the output picture when alpha blending is enabled.

[14] Again neglecting the extra delay due to alpha blending.

Chapter 6

Floating point co-processor

6.1 Purpose

Patches define floating-point equations that are evaluated at each vertex (subsection 2.1.2). Furthermore, per-frame variables such as `zoom`, `rot` or `cx` alter the texture coordinates at each vertex, and correspond to built-in per-vertex equations.

We would like to be able to generate a mesh of up to 64x64 vertices at 30 frames per second, that is to say, compute the X and Y texture coordinates for 122880 vertices each second. With a 100MHz system clock, we have, on average, 813 cycles to fully process each vertex. We want to be able to do 470 basic floating point operations (addition, subtraction, multiplication) per vertex.[1] This means we have, on average, 1.73 cycles to perform each basic floating point operation.

This rules out any software-based implementation. Even assuming a favorable case where patches are compiled (not interpreted) and the LatticeMico32 ISA (section 7.1) equipped with a traditional floating point unit, each basic floating point operation would take approximately 5 cycles at least.[2] Even at 100% CPU utilization, a software implementation would be 3 times too slow!

We have therefore designed and implemented a co-processor for those computations, called PFPU (Programmable Floating Point Unit).

6.2 Forms of parallelism

We need a parallel implementation to solve the problem of performance. Two approaches can be thought of: *vertex-level parallelism* and *instruction-level parallelism*.

[1] The MilkDrop patches contain many other functions. We count divisions and square roots as 15 basic floating point operations and trigonometric functions as 10. We want a patch to be able to do, per vertex, 150 base operations, 8 divisions/square root extractions, and 20 trigonometric operations. This yields the estimate of $150 + 8 \cdot 15 + 20 \cdot 10 = 470$ basic operations.

[2] FPGAs are unlikely to compute the totality of a floating point operation in less than 50ns, which is 5 cycles at 100MHz. Since LatticeMico32 uses a fixed-length in-order pipeline, the CPU would need to be stalled during those cycles.

Vertex-level parallelism is a very simple concept. Since the vertices are independent, the idea is to process several at once. The problem with this approach is that a significant amount of (typically on-chip) memory is needed to store all the intermediate results generated by this technique.

Instruction-level parallelism consists in carrying out in parallel two or more independent computations *for the same vertex*. With this approach, the vertices can be computed one by one, which reduces the required amount of on-chip storage for intermediate results and simplifies the memory access subsystem (as it does not have to handle accesses to different vertices stemming from the same PFPU program).

The two approaches are not mutually exclusive. A hybrid solution can consist in starting with instruction-level parallelism, and then try to schedule more vertices into the same program until the on-chip memory capacity is exceeded.

We do not want such a complex solution to begin with, so we are going with instruction-level parallelism only. If more performance is needed, the addition of vertex-level parallelism can be tried at the expense of relatively small modifications to the hardware.

6.3 Hardware architecture

6.3.1 Overview

A fully software-based extraction of instruction-level parallelism was chosen for several reasons:

- Hardware-based extraction is more costly in terms of resources and more difficult to develop.

- In terms of performance, choosing an hardware-based extraction only pays off compared to a software-based solution when some delays can only be determined at run-time (for example, the memory references). In our case, computations last for approximately 800 cycles and end up with a memory write phase that takes approximately 10 cycles, so the memory delays are negligible and all the other processes (arithmetic pipelines and register writes) have known latencies. Furthermore, we only write to the memory, which means that we can even use a write queue to completely hide the memory write latency.

- A software-based instruction scheduler can have a large instruction window, and thus extract more parallelism than an hardware solution could.

This static scheduling technique makes the floating point co-processor close to a VLIW (Very Long Instruction Word) processor — even though its instruction words are, in fact, not particularly long, as the design is very simple (only one operation is issued per instruction, and jumps are not supported). The architecture we designed is outlined in figure 6.1.

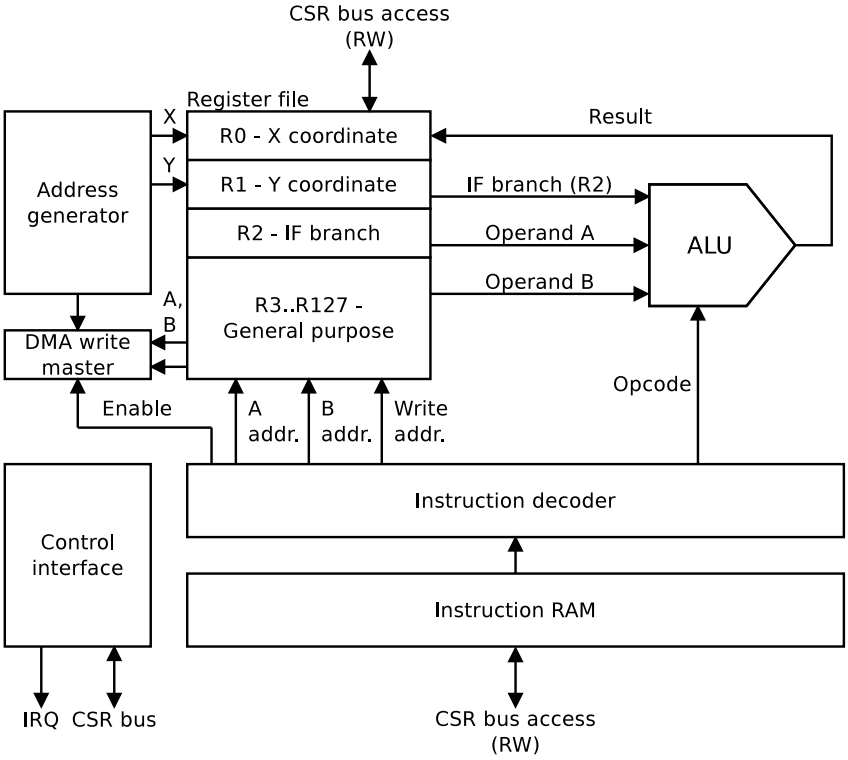

Figure 6.1. Hardware architecture of the floating point co-processor.

Parameter	Operand A	Operand B	Opcode	Destination
Length	7	7	4	7
Bits	`24..18`	`17..11`	`10..7`	`7..0`

Table 6.1. PFPU instruction format.

6.3.2 Instruction set

The 25-bit PFPU instruction format is given in table 6.1. The PFPU executes one such instruction per clock cycle.

An instruction can be split into two parts: the *issue* part, made of the two operand fields and the opcode field, and the *commit* part, made of the destination field. At each cycle, the issue part can fetch two operands from the register file and pushes them into one of the several arithmetic pipelines forming the ALU (Arithmetic and Logic Unit), selected by the "opcode" field. At the same time, the commit part can fetch one result from the ALU (stemming from a previously issued operation that has just finished) and write it back to the register file.

The PFPU program must be written so that all data dependencies are satisfied and only up to one instruction finishes at any given clock cycle (having more than one would create a conflict as there is only a single write port on the register file).

A special instruction is used to write the final result to the memory. It writes the two operands in a format that can be directly read as a texture coordinate by the texture mapping unit (chapter 5). Upon the execution of this instruction, the program for the current vertex is terminated and run again for the next, until all vertices have been processed.

6.3.3 Instruction RAM

The instruction RAM belongs on-chip for two simple reasons:

- it is small: it must only contain hundreds of instructions (the program for one vertex), so its capacity is only a few kilobytes.
- it transfers a large amount of data: one 25-bit instruction on each clock cycle at 100MHz yields a bandwidth of 2.5Gb/s.

There are no jumps or loop structures (for each vertex, the program is always executed linearly), so it does not make sense to replace it with a DRAM-backed cache.

6.3.4 ALU

Overview

The Arithmetic and Logic Unit (ALU) uses 32-bit floats using the same representation as specified by the IEEE 754 standard. This gives enough precision for graphics operations.

The major pipelines of the ALU are listed below.

Basic operations

The unit has pipelines that perform common operations: addition, subtraction, multiplication, absolute value and conversion between floats and two's complement integers.

Inverse square root approximation

The ALU can compute an approximate of the inverse square root of a floating point number using the Quake III method [18]. The ALU only performs the integer operation `0x5f3759df - (i >> 1)`, the subsequent floating point Newton-Raphson iterations needing to be done with additional instructions. This is described in subsection 6.4.1.

Sine and cosine

Sine and cosine are implemented with a look-up table and some logic that exploits the evenness and periodicity of those functions to reduce the size of the look-up table. To compute the sine or cosine of a floating point number, additional instructions need to be generated to convert this number into an integer suitable for indexing the look-up table.

Comparisons

The ALU can test the equality of two floating point numbers and test if one is greater than the other, using two separate opcodes. The result of these operations is 0.0 or 1.0, which can be used as a floating point number in other computations or written to register R2 to implement a conditional statement.

Conditional statements

Conditional statements (if... then... else...) are relatively uncommon in MilkDrop patches, so a low area and straightforward — but slow — implementation was chosen. A special opcode enables a multiplexer that switches between operand A and B and returns the result. The multiplexer is controlled by the register R2, the value of this register being null or non-null selects one or the other input.

Thus, to implement a conditional statement, the PFPU would need to compute both of its branches and store their values in registers (including the branch that is not taken), evaluate the condition and store its value in R2 and finally execute an `IF` operation.

6.4 Run-time compiler

Equations from the patches need to be compiled and scheduled before they can be evaluated by the programmable floating point unit (PFPU). These operations take place on the CPU core of the SoC (section 7.1).

6.4.1 Compilation into virtual machine instructions

The first step in this process is the compilation proper, i.e. parsing the equations and generating instructions for the so-called *floating point virtual machine* (FPVM). This virtual machine has the following properties:

- It has the same operations and opcodes as the PFPU.
- Instructions complete and write their result to the register file in one cycle.
- The number of registers is unlimited.[3]

The compiler employs the fact that the number of registers is unlimited to generate a code that is free from false and output dependencies[4] (by allocating a new register for each result) in order to simplify the task of the scheduler (subsection 6.4.2), which does not have to perform register renaming to create more opportunities for out-of-order execution.

Because of the limited functionality of the operations supported by the FPVM (and the PFPU), some "compound" functions need to be implemented with several instructions. They are detailed below.

Sine and cosine

The sine and cosine instructions expect an integer angle expressed in $\frac{1}{8192}$ turns. Angles outside of the range $[0, 8191]$ are correctly processed (i.e. multiples of 8192 are added or subtracted, by simply ignoring the most significant bits of the input). Therefore, to return the sine or cosine of a floating point angle expressed in radians, the compiler needs to generate three instructions:

1. Multiply by $\frac{8192}{2 \cdot \pi}$.
2. Convert to integer.
3. Look-up the sine or cosine value.

Inverse square root

The inverse square root ($\frac{1}{\sqrt{x}}$) is implemented using the Quake III algorithm [18], reproduced in figure 6.2 with the two Newton-Raphson iterations for improved precision. The input of the algorithm is x and the output y. The cast_to_float function

[3] Actually, it is limited to 2^{32} in our implementation, which can be considered unlimited for our purposes.

[4] Almost. In order to interface in a simple way the code with the "outside world" (subsection 6.4.3), the compiler can be constrained to allocate a given variable to a given register. Depending on how the variable is used, false and output dependencies might actually appear. The scheduler therefore has to check for write-after-read (WAR) and write-after-write (WAW) hazards. Furthermore, conditional statements can cause WAR and WAW hazards around the R2 register, since the IF operation can only use R2 to get the condition value from. However, since all these hazards are extremely rare, they can be resolved by waiting (instead of register renaming) without a significant impact on performance.

$y \leftarrow \text{cast\_to\_float}(0\text{x}5\text{f}3759\text{df} - (\text{cast\_to\_integer}(x) >> 1))$
$y \leftarrow y \cdot (1.5 - 0.5 \cdot x \cdot y \cdot y)$ // first Newton-Raphson iteration
$y \leftarrow y \cdot (1.5 - 0.5 \cdot x \cdot y \cdot y)$ // second iteration

Figure 6.2. Fast inverse square root algorithm.

returns the float whose binary representation as specified by IEEE 754 is the same as that of the integer parameter. The cast_to_integer function performs the opposite operation. They allow the bit twiddling of the numbers in order to produce an initial approximation of the result, which is then refined using the Newton-Raphson method. This is the core idea of the algorithm.

The algorithm produces an approximate value of the inverse square root, with a worst case relative precision of $4.66 \cdot 10^{-6}$ over all floating point values when the two Newton-Raphson iterations are used (according to [18]).

Only a special instruction for performing the first approximation step is provided. The compiler must therefore generate extra multiplication and subtraction instructions for the two iteration steps.

Square root

The square root is implemented using inverse square root with an additional multiplication, according to $\sqrt{x} = x \cdot \frac{1}{\sqrt{x}}$. The relative precision stays approximately[5] the same.

Inverse and division

The inverse of positive numbers is implemented by squaring the inverse square root: $\frac{1}{x} = \frac{1}{\sqrt{x}} \cdot \frac{1}{\sqrt{x}}$. The relative precision obtained is approximately $9.32 \cdot 10^{-6}$.

Divisions of arbitrary numbers are performed by first transferring the sign of the denominator to the numerator, and then multiplying it by the inverse (obtained as above) of the denominator: $\frac{a}{b} = \text{sign}(b) \cdot a \cdot \frac{1}{\sqrt{|b|}} \cdot \frac{1}{\sqrt{|b|}}$. The relative precision is still approximately $9.32 \cdot 10^{-6}$.

The compiler needs to generate many instructions to implement a division, which makes it a rather slow process. However, with this method, very little hardware needs to be added to the PFPU: only support for transferring the sign to the numerator needs to be implemented.

6.4.2 Scheduling

Once the complete code is available as FPVM instructions, the next step is to map these instructions to the PFPU and schedule them.

[5] Not counting the loss of precision incurred by truncating the mantissa of the floating point multiplication result.

Our scheduling algorithm proceeds cycle by cycle. At each PFPU cycle (which corresponds to a PFPU instruction), it searches the complete set of FPVM instructions waiting to be scheduled for one that meets the following four conditions:

1. No read-after-write (RAW) hazard. The operands for the instruction to be scheduled must have been stored into the register file of the PFPU, otherwise, the instruction is not scheduled.

2. No write-after-write (WAW) hazard. If there is a previous uncompleted instruction that writes to the same register as the instruction to be scheduled, the instruction is not scheduled.

3. No write-after-read (WAR) hazard. If there is a previous unscheduled FPVM instruction that reads the register that the instruction to be scheduled modifies, the instruction is not scheduled.

4. No output conflict. The instruction to be scheduled must not complete at the same cycle as another previously scheduled instruction.

If an instruction is found, the FPVM registers of its operands are translated to PFPU registers, a PFPU register is allocated for its output register so that further instructions needing the result can read it, and the instruction is written to the PFPU program. If no further instruction needs to read the operands of the instruction just scheduled, the registers are deallocated in order to save the limited PFPU register space (unless they are bound to a constant or user variable, see subsection 6.4.3). The algorithm then proceeds to the next cycle.

It can be noted that the handling of WAW and WAR hazards is zealous, as some hazards detected by the algorithm may actually not be present because of the pipeline delays. Since those hazards are infrequent, this approach does not have a large impact on the performance but simplifies the algorithm.

Several instructions can sometimes meet the conditions to be potentially scheduled at the same cycle. In this case, the algorithm chooses the first one to appear in the FPVM instruction flow. This *greedy* approach can certainly be optimized, though no effort has been made in this direction.

6.4.3 Constants and user variables

Constant values are assigned a specific register by the compiler, that needs to be initialized before the code is run. User variables can also be bound to a given register, and thereby can be read and written from and to the PFPU. Those registers will then be used exclusively for the constant or user variable.

To differentiate registers used by constants and user variables from registers used to store internal results, the former are given positive numbers by the FPVM compiler while the latter are given negative numbers.

Patch	Instructions	Cycles	CPI
Default	192	259	1.35
Fvese - The Tunnel (Final Stage Mix) (simplified)	208	286	1.38
Geiss - Warp of Dali 1	220	292	1.33
Krash - Digital Flame (simplified)	216	293	1.36
Unchained & Rovastar - Wormhole Pillars (Hall of Shadows mix)	231	326	1.41

Table 6.2. Greedy PFPU scheduler performance with the per-vertex math of different MilkDrop patches (Milkymist 0.5.1).

The scheduler then maps all positive FPVM registers to the PFPU register with the same number, so that they can be easily accessed by the user. Negative FPVM registers are dynamically allocated during the scheduling among the remaining PFPU registers.

6.5 Results

The performance of the PFPU depends directly on the performance of the scheduler, that is to say, its ability to take advantage of out-of-order execution opportunities to hide the latencies of the hardware.

We compiled and scheduled a few MilkDrop patches, and counted the resulting number of instructions and the number of cycles after scheduling. The ratio between the two figures is the CPI (Cycles Per Instruction), which represents the average time it takes to execute one instruction. The results are given in table 6.2. The "Default" patch is a patch that contains no user-defined per-vertex equations, and the instructions correspond to the implicit equations needed to implement the built-in effects (zoom, scaling, ...).

From this table, it is apparent that our approach, albeit simple, is very efficient at extracting instruction-level parallelism. Indeed, the CPI stays between 1.38 and 1.41 while the latencies of the hardware pipelines executing the instructions are much higher, between 2 and 5 (table 6.3).

To put this in contrast with our initial goals (section 6.1), let us consider a typical patch that performs, per vertex, 150 base operations (addition, subtraction, multiplication), 4 divisions, 4 square root extractions and 20 sine or cosine computations (which is close to the patch used in our initial estimate). According to the table 6.4, this would mean 318 PFPU instructions. The maximum CPI that lets us achieve our performance goal is therefore 2.56. Our performance goal is easily met, assuming that other patches expose the same opportunities for out-of-order execution.

Instruction	Latency
Floating point addition	4
Floating point subtraction	4
Floating point multiplication	5
Floating point absolute value	2
Conversion from float to integer	2
Conversion from integer to float	3
Sine/cosine table look-up	4
Comparisons	2
Copy	2
Conditional	2
Inversion of the sign of operand 1 if operand 2 is negative	2
Inverse square root approximation	2

Table 6.3. PFPU latencies in cycles (Milkymist 0.5.1).

Operation	Instructions
Addition, subtraction, multiplication	1
Sine and cosine	3
Inverse square root	11
Square root	12
Division	15

Table 6.4. Exact cost in instructions of common operations on the PFPU.

Chapter 7

Software

7.1 LatticeMico32

The heart of the software execution capabilities of the SoC is the LatticeMico32 microprocessor core [30]. It is a classic 6-stage in-order pipelined RISC processor (figure 7.1) with a custom instruction set supported by the GNU (GCC-based) compiler tool chain. It supports separate instruction and data caches with up to two ways. There are an optional barrel shifter, pipelined multiplier and multi-cycle divider.

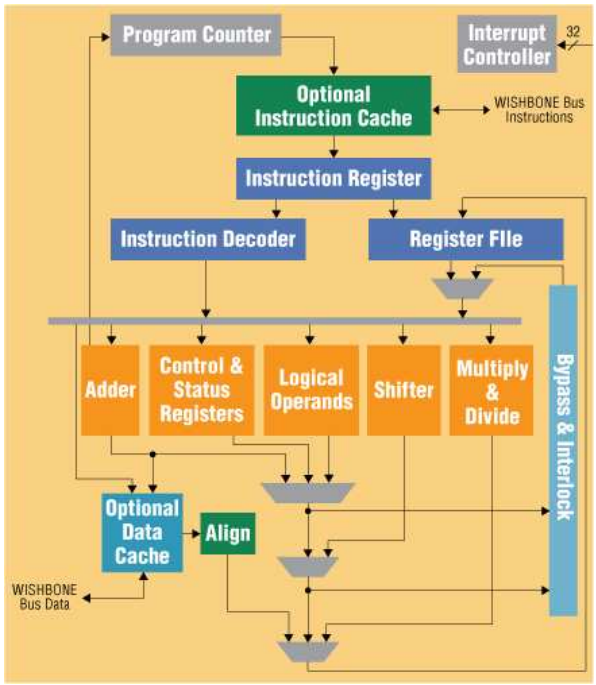

Figure 7.1. LatticeMico32 architecture (Lattice Semiconductor).

The Milkymist system-on-chip uses LatticeMico32 with 2-way caches of 16kB each, and all the optional features enabled.

At the time this thesis is written, LatticeMico32 is the only hardware component that we have not developed specifically for the Milkymist project.

7.2 Capabilities

The "nommu" version of Linux has been ported to the Milkymist SoC (figure 7.2). Since this is a community effort with a significant contribution by Takeshi Matsuya from Keio University, the details are not covered in this Master's thesis. Still, this demonstrates the ability of the platform to run complex software.

Figure 7.2. Linux booting on the Milkymist SoC.

7.3 Benchmarking

The performance of the Milkymist SoC was compared to Microblaze [34], the proprietary Xilinx soft-core SoC platform.

The benchmark used was the "consumer" MiBench [14] suite. By contrast to traditional benchmarks such as SPEC, MiBench is tailored to typical workloads of embedded systems. Only two benchmarks are missing from the "consumer" set: `tiff2rgba` (it tried to use too much contiguous memory for the nommu Milkymist/Linux allocator to handle) and `lame` (it crashed on Microblaze).

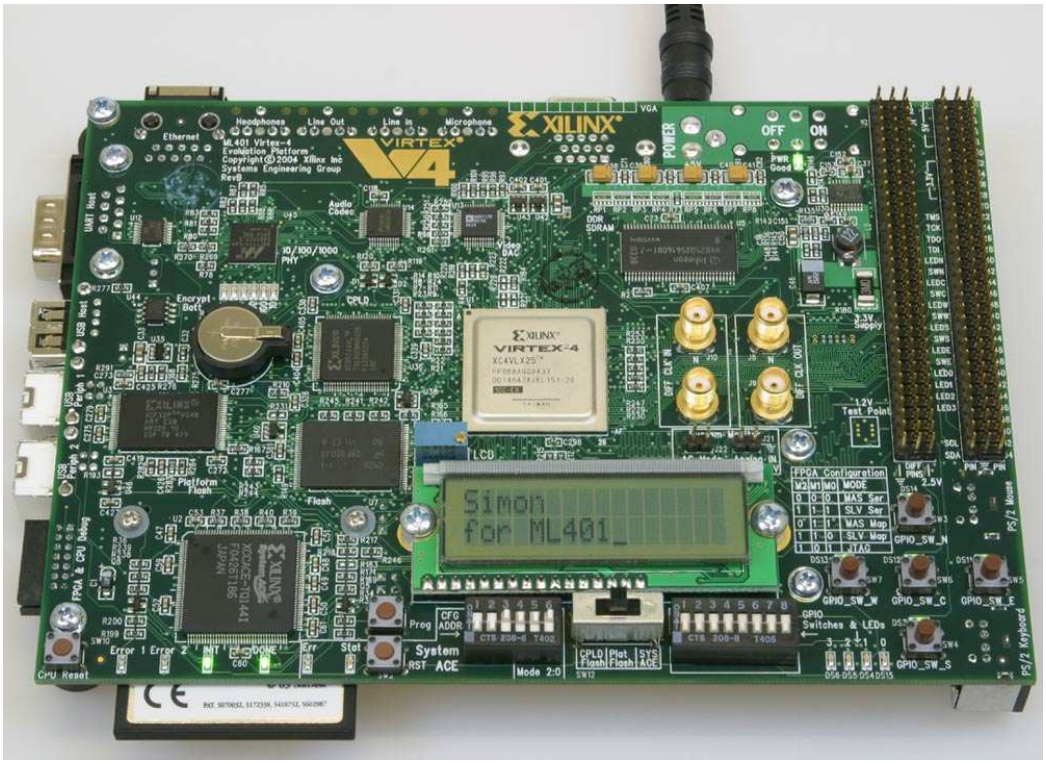

Figure 7.3. Xilinx ML401 development board.

All tests were run on a Xilinx ML401 (XC4VLX25 FPGA, see figure 7.3) development board, with a system frequency of 100MHz.

For Milkymist, the configuration used was the default one of the port to the ML401 board:

- Processor with hardware multiplier, divider and barrel shifter
- 16kB L1 instruction and data (write-through) cache (2-way set-associative)
- No memory management unit (LatticeMico32 does not have one)
- 16kB FML bridge write-back L2 cache (direct mapped)
- HPDMC DDR SDRAM controller, 32-bit SDRAM bus width
- 100MHz DDR SDRAM clock

Benchmark	Run 1	Run 2	Average	Deviation
jpeg	2.57 s	2.54 s	2.56 s	1.18 %
mad	5.84 s	5.87 s	5.86 s	0.51 %
tiff2bw	9.51 s	9.69 s	9.6 s	1.89 %
tiffdither	19.28 s	19.3 s	19.29 s	0.10 %
tiffmedian	26.48 s	26.26 s	26.37 s	0.84 %
typeset	21.44 s	21.79 s	21.62 s	1.63 %

Table 7.1. User execution times on Milkymist 0.2.

- Video output running at standard VGA resolution, consuming approximately 300MBps of memory bandwidth
- Software: GCC 4.2.1 and Linux 2.6.23.

For Microblaze, the configuration is as follows:

- Processor with hardware multiplier, divider and barrel shifter
- 16kB L1 instruction and data (write-through) cache (direct mapped, multi-way caches are not supported)
- Full memory management unit
- No L2 cache (not supported)
- MPMC DDR SDRAM controller, 32-bit SDRAM bus width
- 100MHz DDR SDRAM clock
- No video output
- Software: GCC 4.1.2 and Linux 2.6.32.4.

The comparison seems clearly in favor of Milkymist, with a rough 15%-35% (depending on the benchmark) reduction in execution time. Details are shown in figure 7.4 and in tables 7.1 and 7.2. Deviation is computed as:

$$\frac{|t_1 - t_2|}{\min(t_1, t_2)} \tag{7.1}$$

It is meant to check that the results are deterministic and reproducible.

The root causes of this performance improvement were not investigated; but since LatticeMico32 and Microblaze share a very close architecture, it is suspected that these differences are vastly explained by the combination of the low-latency HPDMC memory controller and the improved caches.

The main point of this comparison is to confirm the viability of Milkymist as a powerful SoC platform, that can withstand the competition with proprietary solutions.

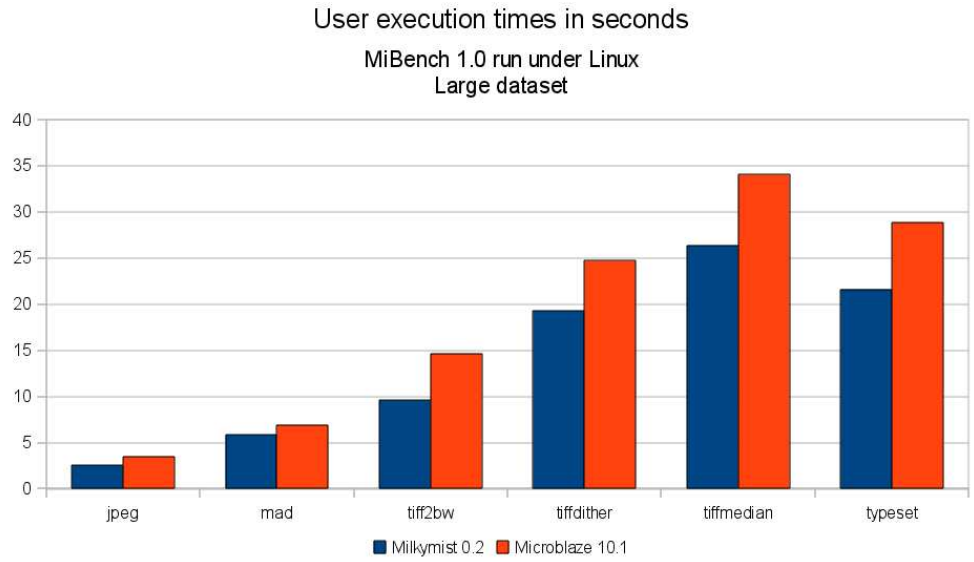

Figure 7.4. Comparative MiBench results of Milkymist and Microblaze.

Benchmark	Run 1	Run 2	Average	Deviation
jpeg	3.42 s	3.58 s	3.5 s	4.68 %
mad	6.72 s	7.11 s	6.92 s	5.80 %
tiff2bw	15.19 s	14.12 s	14.66 s	7.58 %
tiffdither	24.72 s	24.68 s	24.7 s	0.16 %
tiffmedian	35.02 s	33.05 s	34.04 s	5.96 %
typeset	28.91 s	28.83 s	28.87 s	0.28 %

Table 7.2. User execution times on Microblaze 10.1.

7.4 Design of a MilkDrop-like rendering program

7.4.1 Description

With all those elements at hand, the last task is to design a complete renderer program compatible with MilkDrop. The block diagram of our renderer is given by the figure 7.5.

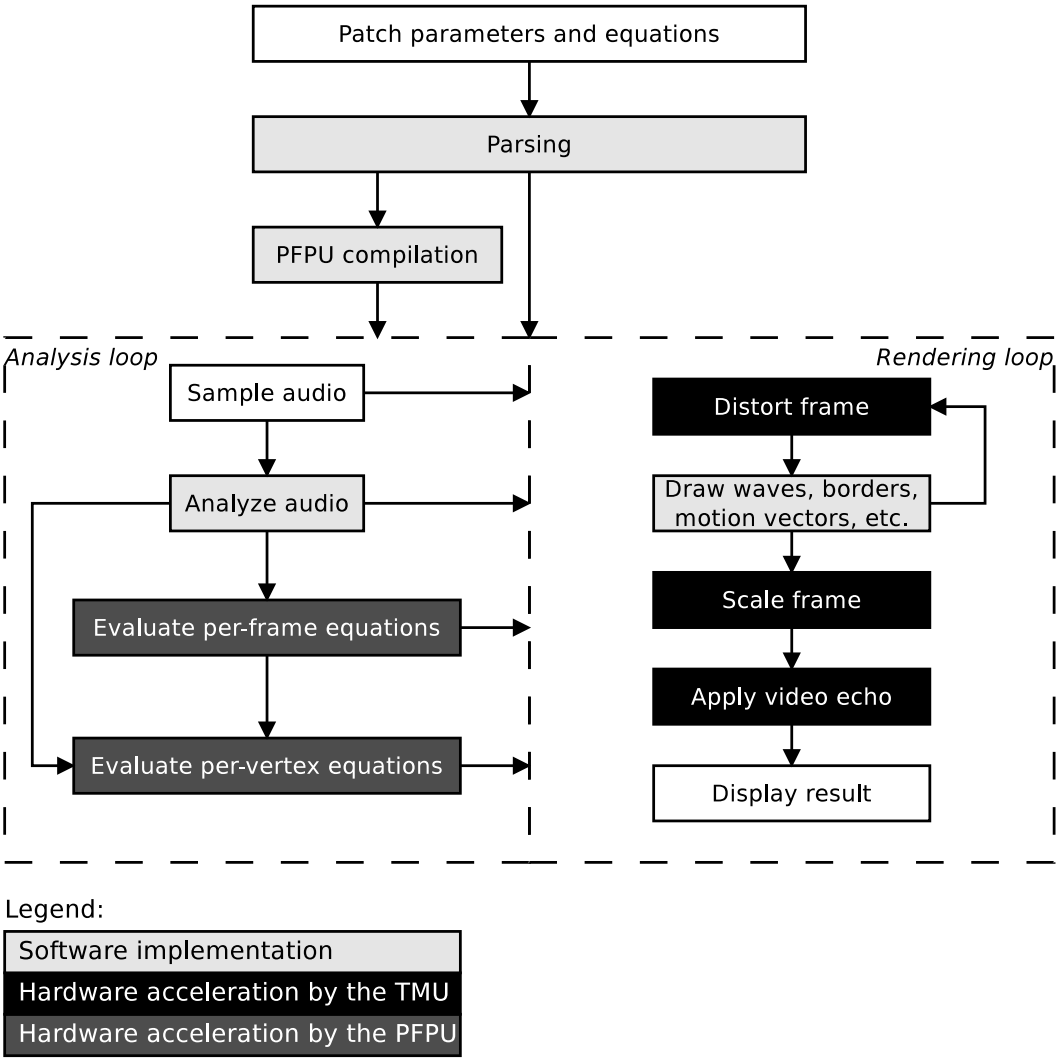

Figure 7.5. Rendering software architecture.

Before the rendering proper can take place, the code of the patch needs to be parsed and microcode for the PFPU generated (section 6.4). This process, implemented entirely in software, is slow (several hundreds of milliseconds) but it is not performance-critical, as it only needs to be done once before the rendering. Further-

more, its results could be cached to allow a smooth transition between pre-defined patches.

The first step of the rendering process is to digitize the audio signal. This is achieved using the system-on-chip's AC97 controller and its device driver. They are programmed to pack the audio samples into buffers, each of them holding exactly the number of samples that corresponds to the desired video frame rate of 30 frames per second. The buffers are written to the memory using DMA through the L2 cache.

The next operation is to analyze each audio buffer to extract its energy content in three frequency bands, in order to generate the `bass`, `mid` and `treb` parameters that can be used in MilkDrop patches to connect the visual effects to sound. This is done using three decimating finite impulse response (FIR) filters, each followed by an accumulator that adds the energies of each sample at its filter's output. This is several times faster than the original MilkDrop implementation, which consisted in performing a Fourier transform followed by a spectral summation of the energies in the three bands, and allows a software implementation. Indeed, this process has been observed to take approximately 8ms when run on the system-on-chip, which is more than 3 times less than the video frame period. The filters operate directly on the audio data sent into the memory by the AC97 controller, avoiding any time-wasting memory copy.

The next step is to evaluate the per-frame equations. Even though hardware acceleration is not really needed there, this step is still performed on the PFPU in order to re-use the existing compilation and evaluation infrastructure.

Once per-frame parameters are known, the renderer proceeds to evaluating per-vertex equations on the PFPU. This generates the full mesh of vertex coordinates to be used later by the texture mapping unit for distorting the frame.

All the processes we have seen so far are part of the *analysis loop*. Its output is a stream of packets, each packet describing the operations to be done for one frame. One packet contains the audio samples, the results of the audio analysis, the outputs of the per-vertex equations (as well as the fixed patch parameters) and the distortion mesh data. The packet is sent to the *rendering loop*.

Upon reception of the packet, the rendering loop takes its current frame, and runs the TMU (chapter 5) to distort it.

Then, it superimposes waves, borders and motion vectors on the result. This is implemented in software, as the processor is fast enough for these tasks.

Finally, the TMU is run twice to scale the internal frame buffer to the screen size and to apply the video echo.

The output is now ready to be displayed. This is done by simply instructing the VGA controller to switch to the newly generated frame buffer. The VGA controller then performs the requested buffer switch during the next vertical blanking interval, in order to produce a transition without any tearing artifact. A triple-buffering technique is used so that the software never has to wait for the VGA controller to

release a buffer,[1] at the expense of an increased memory consumption.

7.4.2 Cache coherency

The system-on-chip provides limited support for cache coherency (section 4.5). Therefore, several operations to ensure coherency must be done by the software throughout the rendering process.

Cache coherency within the analysis loop

The only precaution that should be taken is to invalidate the L1 cache after each received buffer from the AC97 audio controller.

There is no need to invalidate the any cache after evaluation of the per-frame equations, as the outputs are read directly from the PFPU register file which is mapped in the non-cache-able CSR address space.

The output the per-vertex equations is sent directly to the rendering loop.

Cache coherency between the analysis and rendering loops

Among the data sent from the analysis loop, there is one element which is not coherent with respect to the L1 cache: the vertex data generated from the per-vertex equations. However, this data is not read by the CPU but only by the TMU. The latter fetches it from the L2 cache, which is also where the PFPU writes data. Therefore, no operation is needed to ensure cache coherency.

Cache coherency within the rendering loop

Most operations are done by the texture mapping unit, which deals directly with the SDRAM. There are two cases where cache coherency issues can arise:

1. Between the CPU and the TMU during the wave (and other elements) drawing process.

2. Between the CPU and the VGA controller. Indeed, the VGA controller makes coherent transactions with respect to the L2 cache. If the cache holds an outdated copy of the data, it will be used instead of the more recent version in SDRAM.

To solve these issues with a minimal number of cache invalidations, we make sure that the L1 and L2 cache never hold any data from any frame buffer except during the wave drawing process. This is ensured by:

[1] Assuming that the frame rate is less than the screen refresh frequency, which is always the case in practice as the frame rate is limited to 30 fps and all display devices refresh at much more than 30 Hz.

1. Aligning all frame buffers to a multiple of both the line lengths of the L1 and L2 caches (i.e. of the least common multiple of the line lengths). In our case, this does not add any additional constraint as those buffers already had tighter alignment requirements stemming from the VGA controller and the avoidance of inter-channel cache conflicts within the texture mapping unit.

2. Invalidating the L1 and L2 cache just after the wave drawing process.

7.4.3 Event-driven operation

Our implementation is *event-driven*. Instead of being fully sequential, the software waits for and acts upon events (for example, the texture mapping unit finishing processing, or a new audio buffer being ready). After an event is received, the CPU can either process the data itself (for example, run the FIR filters after a new audio buffer is ready) or run another hardware unit (for example, run the TMU after the PFPU has evaluated the per-vertex equations). This approach improves performance by letting the three main processing units of the system (the CPU, the PFPU and the TMU) operate in parallel.

7.4.4 Results

Our system is able to render successfully many original MilkDrop presets at 30 frames per seconds in 640x480 resolution. However, it is often operating near its limit, and sometimes above it, as some presets exhibit a lower frame rate. This is often due to the fact that drawing the waves and the borders take a long time on the CPU, especially when many semi-transparent pixels need to be drawn. This precludes the possibility of supporting presets that employ complex custom waves and shapes (that would be drawn by the CPU), unless further acceleration techniques are developed.

Chapter 8

Conclusion and future works

Through this thesis project, we have covered many different aspects of computer architecture, which were necessary to achieve our goal of designing a high-performance system-on-chip for video synthesis.

We first exposed the difficulties involved with the amount of memory required for the video synthesis application, and the latency and bandwidth challenges stemming from the DRAM technology. Our solution consisted of a combination of a page mode control algorithm, using of burst transfers with critical-word-first support, and pipelining. It keeps the hardware small and simple, and our results have shown that it allows using the memory bandwidth at roughly half the peak capacity of the chips, which was enough for our application taking into account an oversizing of the memory system.

Then, we explained how we split the system interconnect on three different busses, solving high fanout and large multiplexer problems, enabling devices on different bus segments to communicate in parallel, and having specific bus standards on each segment, depending on the feature and bandwidth needs of the devices using it.

We went on with the problems stemming from the compute-intensive parts of the video rendering process: texture mapping and fast evaluation of the equations that define the texture coordinates. We solved those by developing custom hardware accelerators, the texture mapping unit (TMU) and the programmable floating point unit (PFPU). The PFPU easily exceeded its design goals, but our TMU was challenged by its important consumption of memory bandwidth and the associated memory latencies issues. It was nonetheless able to meet our expectation of enabling the rendering of the video effects in VGA (640x480) resolution.

Finally, we chose and integrated a good CPU core to control the system and perform less compute-intensive and software-friendly tasks. Our choice was the LatticeMico32 core, which, when integrated into our system, exceeded the performance of the competing proprietary Microblaze solution. We wrote video rendering software for it, leveraging the possibilities of our SoC architecture.

Overall, our goal has been met as we have been successful at rendering many MilkDrop presets in VGA resolution on our system at a good frame rate. However, several tracks for computer architecture related improvements can be thought of:

1. In order to be able to use more memory bandwidth from the SDRAM chips, an out-of-order memory controller could be designed.

2. The texture mapping unit could use a prefetching technique to be less affected by the memory latency. Such a technique could also enable several outstanding memory requests from the texture mapping unit at the same time, allowing the memory controller to reorder them in order to leverage more bandwidth from the SDRAM chips.

3. During the development of the texture mapping unit (which was done in plain Verilog HDL), we felt that it was not very productive to repeatedly design manually the pipeline interlocking logic for each stage. This made us think of an ambitious research project that could consist in designing a programming language that would describe similar pipelines from a higher level of abstraction, which would bring many advantages. First, productivity would be improved as the designer would not have to design and code manually (with a risk of errors) many elements. Second, the language could be simulated to validate the high level functionality of the design. Third, this simulation could also be used to explore the design space of functionally-equivalent implementations with different area, power and speed performances (for example, by observing the impact on speed that the cache size of a memory access point has in order to strike a good compromise between the two). A computer program could even be used to perform part or all of this exploration in order to meet pre-defined power, area and speed goals, and back-annotate all the design choices it made into the design.

   With such a powerful tool, we could, for example, quite easily upgrade the texture mapping unit's graphics pipeline so it could support the full OpenGL ES specification. It could also certainly be used in many other fields unrelated to computer graphics.

The Milkymist project is not entirely about computer architecture and system-on-chip design. We are also working, in collaboration with Sharism at Work Ltd. and other contributors, on building complete "open source hardware" products around the SoC described herein. Our first device will be the Milkymist One interactive VJ station. Technical aspects of this wider project also include printed circuit board layout (figure 8.1) and software engineering. All of the work is covered by open source licenses.

We hope that this open hardware platform will be successful — used not only for its intended live video synthesis purpose, but also as a learning tool, as a development platform and as a base or even a design library for other open source projects.

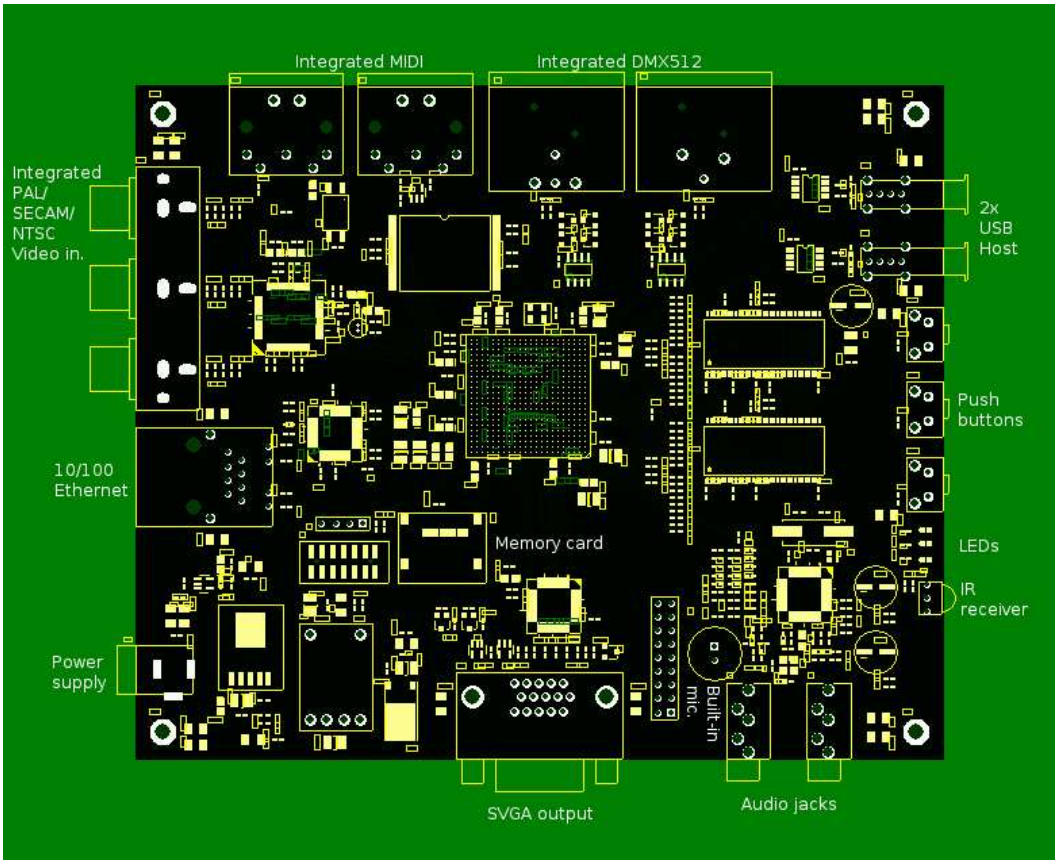

Figure 8.1. Printed circuit board floor plan of the Milkymist One.

Bibliography

[1] M. Abrash. *Michael Abrash's Graphics Programming Black Book (Special Edition)*. Coriolis Group Books, 1997.

[2] G. Allan. DDR SDRAM: A low cost, yet increasingly complex off-chip memory solution for SoCs. `https://www.synopsys.com/dw/doc.php/wp/ddr_sdram_wp.pdf` (Retrieved on 22/04/2010), 2007.

[3] Altera. Nios II performance benchmarks. `http://www.altera.com/literature/ds/ds_nios2_perf.pdf` (version 5.0, retrieved on 30/04/2010).

[4] Altera. Nios II processor: The world's most versatile embedded processor. `http://www.altera.com/products/ip/processors/nios2/ni2-index.html` (Retrieved on 21/04/2010).

[5] Arkaos. GrandVJ. `http://www.arkaos.net` (Retrieved on 21/04/2010).

[6] S. Bourdeauducq. Milkymist interactive VJ station. `http://www.milkymist.org` (Retrieved on 21/04/2010).

[7] S. Bourdeauducq. Configuration and status register (CSR) bus specifications. `http://www.milkymist.org/doc/csr.pdf` (Retrieved on 21/04/2010), 2009.

[8] S. Bourdeauducq. FastMemoryLink (FML) bus specifications. `http://www.milkymist.org/doc/fml.pdf` (Retrieved on 21/04/2010), 2009.

[9] Silicore Corporation and OpenCores.org. WISHBONE System-on-Chip (SoC) interconnection architecture for portable IP cores. `http://opencores.org/downloads/wbspec_b3.pdf` (Retrieved on 21/04/2010), 2002.

[10] C. Dawson, S.K. Pattanam, and D. Roberts. The Verilog procedural interface for the Verilog hardware description language. In *Verilog HDL Conference, 1996. Proceedings.*, pages 17–23, Santa Clara, CA, USA, 1996. IEEE International.

[11] E. de Koning and B. van der Ploeg. Resolume. `http://www.resolume.com` (Retrieved on 21/04/2010).

[12] N. Feske and M. Alles. Genode FX: an FPGA-based GUI with bounded output latency and guaranteed responsiveness to user input. `http://www.genode-labs.com/ publications/ genode-fpga-graphics-2009.pdf` (Retrieved on 22/04/2010).

[13] Aeroflex Gaisler. SoC library. `http://www.gaisler.com` (Retrieved on 21/04/2010).

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[15] S. Heithecker and R. Ernst. Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*, pages 575–578, New York, NY, USA, 2005. ACM.

[16] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a texture cache architecture. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 133–ff., New York, NY, USA, 1998. ACM.

[17] Sonics Inc. MemMax scheduler. `http://www.sonicsinc.com/uploads/pdfs/ MMScheduler_ds_final02_032109.pdf` (Retrieved on 21/04/2010).

[18] Chris Lomont. Fast inverse square root. `http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf` (Retrieved on 07/05/2010).

[19] M. Lu. *Arithmetic and logic in computer systems.* John Wiley and Sons, 2004.

[20] D. Magdic. Visikord. `http://visikord.com/` (Retrieved on 22/04/2010).

[21] D. Magdic. Wiimote-reactive MilkDrop visuals. `http://forums.winamp.com/showthread.php?threadid=289007` (Retrieved on 22/04/2010).

[22] D. Mattson and M. Christensson. Evaluation of synthesizable CPU cores. Master's thesis, Göteborg, Sweden, 2004.

[23] Sun Microsystems. OpenSPARC. `http://www.opensparc.net` (Retrieved on 21/04/2010).

[24] W. Miller. Real word applications for field programmable gate array devices – an overview. In *WESCON/94. 'Idea/Microelectronics'. Conference Record*, pages 548–551, Anaheim, CA, USA, 1994. IEEE.

[25] Nullsoft. Milkdrop plug-in for Winamp. `http://www.nullsoft.com/free/milkdrop/` (Retrieved on 21/04/2010).

[26] Opencores. OpenRISC. `http://www.opencores.org/project,or1k` (Retrieved on 21/04/2010).

[27] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM. *IEEE Micro*, 17:34–44, 1997.

[28] Simply RISC. Simply RISC S1 core. `http://www.srisc.com/?s1` (Retrieved on 21/04/2010).

[29] T. Rokicki. Indexing memory banks to maximize page mode hit percentage and minimize memory latency. Technical report, HP Laboratories Palo Alto, 2003. `http://www.hpl.hp.com/techreports/96/HPL-96-95R1.html` (Retrieved on 21/04/2010).

[30] Lattice Semiconductor. LatticeMico32. `http://www.latticesemi.com/products/intellectualproperty/ipcores/mico32/index.cfm` (Retrieved on 21/04/2010).

[31] Pragmatic C Software. GPL Cver. `http://sourceforge.net/projects/gplcver/` (Retrieved on 04/05/2010).

[32] Wikipedia. MilkDrop. `http://en.wikipedia.org/wiki/MilkDrop` (Retrieved on 21/04/2010).

[33] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[34] Xilinx. Microblaze soft processor core. `http://www.xilinx.com/tools/microblaze.htm` (Retrieved on 21/04/2010).