2nd FPGAworld CONFERENCE

SEPTEMBER 8, 2005
ELECTRUM-KISTA, STOCKHOLM, SWEDEN

Editors:
Lennart Lindh and Vincent J. Mooney III

PROCEEDINGS 2005

The FPGAworld Conference addresses all aspects of digital and hardware/software system engineering on FPGA technology. It is a discussion and network forum for researchers and engineers working on industrial and research projects, state-of-the-art investigations, development and applications. The proceedings only contain the academic presentations; for the industrial presentations see (www.fpgaworld.com/conference).

ISSN 1404-3041 ISRN MDH-MRTC-188/2005-1-SE

SPONSORS

Additional copies of 2005 or prior Proceedings may be found at www.FPGAworld.com
or at Malardalens University library (www.mdh.se), ISSN 1404-3041 ISRN MDH-MRTC-188/2005-1-SE

2005 PROGRAM COMMITTEE

Academic Program Chairman

Professor Vincent J. Mooney III, Georgia Institute of Technology, USA

Academic Program Committee Members

Solfrid Sjåstad Hasund, Bergen University College, Norway
Lennart Lindh, Mälardalen University, Sweden
Adam Postula, University of Queensland, Australia
Pramote Kuacharoen, National Institute of Development Administration, Thailand
Mohamed A. Shalan, Ain Shams University, Egypt

Industrial Program Chairman

Lennart Lindh, Mälardalen University, Sweden

Industrial Program Committee Members

Kim Petersén, HDC, Sweden
Mickael Unnebäck, OpenCores, Sweden
Fredrik Lång, EBV, Sweden
Niclas Jansson, BitSim, Sweden
Göran Bilski, Xilinx, Sweden
Adam Edström, Elektroniktidningen, Sweden
Kristina Kristoffersson, Arrow, Sweden
Espen Tallaksen, Digitas and Embla, Norway
Göran Rosén, Actel, Sweden
Tommy Klevin, ÅF, Sweden
Tryggve Mathiesen, XDIN, Sweden

Publicity Chairman

David Kallberg, FPGAworld, Sweden

Registration and Local Arrangements

Kerstin Åberg, FPGAworld, Sweden

General Chairman

Lennart Lindh, FPGAworld, Sweden

General Chairman's Message

The FPGAworld program committee welcomes you to the 2nd FPGAworld conference. This year's conference is held in Electrum-Kista, Stockholm, Sweden. We hope that the conference provides you with much more then you expected. This year it is the first time we have academic reviewed papers; this is an important step to incorporate the academic community into the conference program.

We will try to balance academic and industrial presentations, exhibits and tutorials to provide a unique chance for our attendants to obtain knowledge from different views. This year we have over 24 exhibits, 14 reviewed industrial papers (12 accepted), 6 academic papers (3 accepted), 7 sponsors and 2 three-hour hands on tutorials.

The FPGAworld 2005 conference is more then two times bigger than the FPGAworld 2004 conference.

Next year the industrial parts of FPGAworld conference will be in Stockholm, Munich and London. The academic FPGAworld conference will only be in Stockholm.

All are welcome to submit industrial/academic papers, exhibits and tutorials to the conference, both from academic and industrial backgrounds. Together we can make the FPGAworld conference exceed even above our best expectations!

Please check out the website (http://fpgaworld.com/conference/) for more information about FPGAworld 2006. In addition, you may contact the following people for more information:

David Källberg (david@fpgaworld.com) or Kerstin Åberg (k@realfast.se).

Lennart Lindh

General Program Chair

09:00	Conference opening, Lennart Lindh, FPGAworld			
09:15 10:00	Keynote Session Faster, Cheaper and Safer FPGA development, Espen Tallaksen, Norway			
10:30 - 12:00	Chair Göran Bilski	Chair Mikael Unnebäck	Chair Kristina Kristoffersson	Session D1 Hands-on Product Workshop Xilinx
	Session A1 Graph Based Physical Synthesis for FPGAs Session A2 FPGA Superclusters for extreme performance Session A3 Industrial experience using Mobius for rapid hardware development	Session B1 Efficient System Design with SOPC Builder and Altera Products Session B2 Model-Based Design of Embedded Signal Processing Systems with Simulink	Session C1 Case study: Image fusion in real-time Session C2 Annapolis Micro Systems Session C3 BitSim	
13:00 - 14:30	Chair Fredrik Lång	Chair Tryggve Mathiesen	Chair Tommy Klevin	Session D2 Hands-on Product Workshop Synplicity
	Session A4 The Debug Challenge for Today's Platform FPGAs Session A5 Video in FPGA Session A6 Managing Your Software Radio Application for SCA Compliance	Session B3 Power Management for FPGA Designers Session B4 A generic RTL approach to implementing multi-channel designs	Session C4 Next generation of highly integrated flash based FPGA's Session C5 Synective Labs Sessions C6 Design Compiler® FPGA - Fastest Path to Prototype	
15:00 - 16:30	Chair Niclas Jansson	Academic Program Chair Vincent J. Mooney III	Chair Göran Rosén	
	Session A7 New Paradigm of standalone RTOS in Hardware for FPGAs Session A8 System-Level Verification of Signal Processing Applications on ASICs and FPGAs	Session B5 Embedded 3D Graphics Core for FPGA-based System-on-Chip Applications * Session B7 RTL Structuring Using a Schematic Based Code Frame Generator for HDLs * Session B8 A Soft Fixed-Point Digital Signal Processor Applied in Power Electronics*	Session C7 FPGA Re-spins are No Longer "Free" Session C8 Silicon Interfaces	

2005 KEYNOTE ADRESS

Faster, Cheaper and Safer FPGA development

Most companies have a major potential for efficiency improvement in developing complex FPGAs. Paradoxically most of these actually believe they are already really efficient…

The keyword is ‘efficient’ reuse. It is very important to differentiate between 'ad hoc reuse' and 'efficient reuse'. Companies mastering efficient reuse have experienced more than 50% reduction in total development time, - far, far better than any ad hoc reuse…

This keynote presentation will discuss these issues - and then introduce the “Efficient Reuse Methodology - Impact Chart”. This will give the audience an overview of the most important reuse methodology elements, how these affect your current and future projects, the investments required and the actual probability of improvements. Special attention will be given to efficient reuse within a single FPGA project.

Espen Tallaksen, Digitas and Embla, Norway
Espen Tallaksen is an independent consultant on FPGA design and verification - with a special focus on efficiency improvement. He has initiated and assisted in implementing structured verification and efficient reuse in several companies.

Espen Tallaksen graduated from University of Glasgow in Scotland in 1987. He has been involved in numerous ASICs and FPGAs - as a designer, architect and project manager. For more than ten years he has also had special responsibilities for efficiency improvements in companies like Philips Semiconductors in Switzerland, and as a technology manager at Nordic VLSI.

Espen Tallaksen is also coordinating an inter company working group on efficiency improvement (Embla) and a new FPGA development initiative in Norway.

TABLE OF ACADEMIC PROCEEDINGS CONTENTS

Paper B5: Embedded 3D Graphics Core for FPGA-based System-on-Chip Applications

This paper presents a 3D graphics accelerator core for an FPGA based system, and illustrates how to build a System-on-Chip containing a Xilinx MicroBlaze soft-core CPU and our 3D graphics accelerator core. The system is capable of running uClinux and hardware accelerated 3D graphics applications such as a VRML viewer.
The 3D graphics core is connected to a PLB 64-bit on-chip bus, and can render graphics into an on-chip tile buffer, which is later copied, using bus-master DMA transfers, to the frame-buffer in external DDR SDRAM memory. This memory is shared between the CPU, the 3D graphics core, and the video display which periodically reads from memory to display the final rendered graphics. The graphics core uses internal scratch-pad memory to reduce its external bandwidth requirement, this is achieved by implementing a tile-based rendering algorithm. Reduced external bandwidth means that the power consumption is reduced as well.
We show how an FPGA based embedded system is capable of most tasks in a single chip solution, without requiring additional CPU or graphics chips.

Author
Hans Holten-Lund, Informatics and Mathematical Modelling, Technical University of Denmark

Paper B7: RTL Structuring Using a Schematic Based Code Frame Generator for HDLs

The paper decribes design and application of a VHDL RTL code frame generator which is the first step in a more advanced project that shall support the RTL-design process with an always up-to-date ("live") block diagram of the system architecture.
Our approach uses Microsoft Visio based schematic entry because it enables a cheap and flexible code generator implementation with large flexibility for different HDL's including SystemC.
I order to demonstrate practical application of this tool we will show the VHDL design of a PCM31 framer core FPGA for telecommunication applications.

Authors
Jörg Lastig, University of Applied Sciences Hamburg, Hamburg, Germany
Jurgen Reichardt, University of Applied Sciences Hamburg, Hamburg, Germany

Paper B8: A Soft Fixed-Point Digital Signal Processor Applied in Power Electronics

This article presents the “DSPuva16” processor, specifically developed for Power Electronics applications. As its name indicates, this is a Digital Signal Processor oriented to calculation: it operates using fixed-point 16-bit numbers extending its precision up to 24 bits. The MAC operations (rD = rD ± rS * rT) are made regularly using only one instruction cycle, that requires four clock cycles. The processor has been physically tested on FPGA working at 40 MHz.
The application of this processor in Power Electronics (distributed generation systems, AC motor control, active filters) is quite natural, because all working magnitudes are limited in range. The use of several processors is also possible: each one would execute a different regulation loop, with working cycles between 5 and 100 microseconds.

Authors
Santiago de Pablo, University of Valldolid
J. Cebrián, University of Valldolid
L. C. Herrero, University of Valldolid
A. B. Rey, Polytechnic University of Cartagena

Embedded 3D Graphics Core for FPGA-based System-on-Chip Applications

Hans Holten-Lund
Informatics and Mathematical Modelling
Technical University of Denmark
hahl@imm.dtu.dk

Abstract

This paper presents a 3D graphics accelerator core for an FPGA based system, and illustrates how to build a System-on-Chip containing a Xilinx MicroBlaze soft-core CPU and our 3D graphics accelerator core. The system is capable of running uClinux and hardware accelerated 3D graphics applications such as a VRML viewer.

The 3D graphics core is connected to a PLB 64-bit on-chip bus, and can render graphics into an on-chip tile buffer, which is later copied, using bus-master DMA transfers, to the frame-buffer in external DDR SDRAM memory. This memory is shared between the CPU, the 3D graphics core, and the video display which periodically reads from memory to display the final rendered graphics. The graphics core uses internal scratch-pad memory to reduce its external bandwidth requirement, this is achieved by implementing a tile-based rendering algorithm. Reduced external bandwidth means that the power consumption is reduced as well.

We show how an FPGA based embedded system is capable of most tasks in a single chip solution, without requiring additional CPU or graphics chips.

1. Introduction

Hardware accelerated 3D graphics is gaining influence in low-cost embedded devices such as GPS navigators, etc. For some applications FPGAs are on the way to replace CPUs, 3D graphics chips and other ASICs with soft-cores located inside the FPGA, giving a reconfigurable System-on-Chip solution. The main reason for this is the fact that FPGA chips are rapidly getting cheaper, while improving time-to-market. Although dedicated graphics ASICs are used for high performance applications such as game consoles, other applications which require moderate performance above that achievable with software rendering may be better served with a soft-core graphics processor in an FPGA. A similar trend can be observed with soft-core CPUs such as the Xilinx MicroBlaze and Altera Nios replacing traditional CPU chips by moving the CPU into the programmable FPGA fabric.

This paper presents a graphics core for such an FPGA based system, and illustrates how to build a simple SoC containing a Xilinx MicroBlaze soft-core CPU and our 3D graphics accelerator core. The system is capable of running uClinux and hardware accelerated 3D graphics applications such as a VRML viewer.

The target for this implementation is the Xilinx Virtex-4 ML401 XC4VLX25 FPGA evaluation platform. This board features 64 MB 32-bit external DDR SDRAM memory. The peak external memory bandwidth is 800 MBytes/sec when operating at 100 MHz. The board also provides an external ADV7125 3x8-bit RGB video DAC which is used for the VGA display.

The 3D graphics core is connected to a PLB 64-bit on-chip bus, and can render graphics into a frame-buffer using bus-master DMA to external DDR SDRAM memory. This memory is shared between the CPU, the 3D graphics core, and the video display which periodically reads from memory to display the final rendered graphics. The graphics core uses internal scratch-pad memory to reduce its external bandwidth requirement, this is achieved by implementing a tile-based rendering algorithm.

We show how an FPGA based embedded system is capable of most tasks in a single chip solution, without requiring additional CPU or graphics chips.

For the embedded application example with a GPS navigator, the GPS signal processing tasks can also be embedded in the FPGA to avoid the need for a dedicated GPS signal processing ASIC.

2. The Hybris Graphics Architecture

The Hybris graphics architecture [6] is scalable from a minimal embedded implementation to larger parallel implementations. The first FPGA implementation of the graphics architecture is presented in [7] where a simpler version of the core is implemented on a Celoxica RC1000PP PCI board using a Xilinx Virtex XCV1000 FPGA. The FPGA is streaming the input data from a host PC via the PCI bus, and outputs the rendered graphics directly to a VGA monitor.

In this context we will look at a new implementation suitable for an embedded System-on-Chip based on a cheap FPGA and low-cost 32-bit wide DDR SDRAM external memory. The FPGA contains an embedded MicroBlaze CPU and on-chip buses which replaces the PC from the earlier PCI-bus based implementation. To keep

the cost low, the external memory is shared as main memory for the CPU, graphics memory for the 3D graphics accelerator and framebuffer memory for the display. This means that we need to have a bandwidth budget for the complete system. Internally in the FPGA all the main memory traffic is also present on a bandwidth matched 64-bit PLB (Processor Local Bus) on-chip bus.

The Hybris graphics architecture uses tile based rendering to allow for scalable parallel implementations. The method gets its name as it divides the screen into rectangular tiles. Each tile is relatively small so it can easily fit into on-chip memory next to the rendering engine. While we can divide the screen into many tiles, e.g. a 640x480 pixel framebuffer can be divided into 20x15 32x32 pixel tiles, we cannot afford to implement 20x15 tile rendering engines running in parallel. If we have one tile engine rendering to a virtual tile, we can render all tiles in sequence. This is also known as *virtual local framebuffer* see [5]. Similar techniques are also used in [8, 4, 9, 3] as well as by GigaPixel, which was acquired by 3dfx and then Nvidia. Some more recent work is [1, 2]. The advantages are that we do not need to access off-chip memory while rendering a tile, that we can use more bits per pixel in the small tile than we would in a global framebuffer without increasing off-chip bandwidth, that we can render multiple tiles in parallel, and that we can render at a higher resolution to implement supersampled anti-aliasing again without increasing off-chip bandwidth. Unfortunately since we must collect and bucket sort input data (triangles) for a tile before rendering it, we need a potentially large input buffer, which is the main drawback of the tile based rendering method. This is not a problem for scenes with a relatively low triangle count.

3. MicroBlaze Soft-Core CPU

The MicroBlaze [10] is a 32-bit soft-core RISC processor from Xilinx, optimized for FPGA implementation. We use the MicroBlaze CPU core mainly because it is well supported by the Xilinx implementation tools (EDK 7.1 and ISE 7.1), allowing us to focus our work on the system-design and hardware accelerator design. In addition to the usual on-chip bus-interfaces, MicroBlaze also provides eight Fast Simplex Links (FSL) which are useful for direct communication with hardware accelerators. For this application we use the MicroBlaze CPU to run the user programs and if necessary also the uClinux operating system.

The MicroBlaze CPU is also used for the front-end of the graphics architecture. This means that 3D transformation and lighting as well as triangle set-up is done in software on the CPU. The new MicroBlaze CPU version 4.0 provides hardware floating point support to help applications such as this, removing the need to convert the program to fixed-point. In a future implementation a hardware floating point vector co-processing module can be connected to the CPU using one of the FSL links. This makes it possible to implement dedicated matrix-vector multiplication and dot product hardware which is useful for accelerating 3D transformation and lighting. Since a FSL core does not need to send anything back to the CPU, the FSL core can be used to send its processed data directly to the graphics processor core or main memory, bypassing the overhead of sending it back through the CPU and its bus interface.

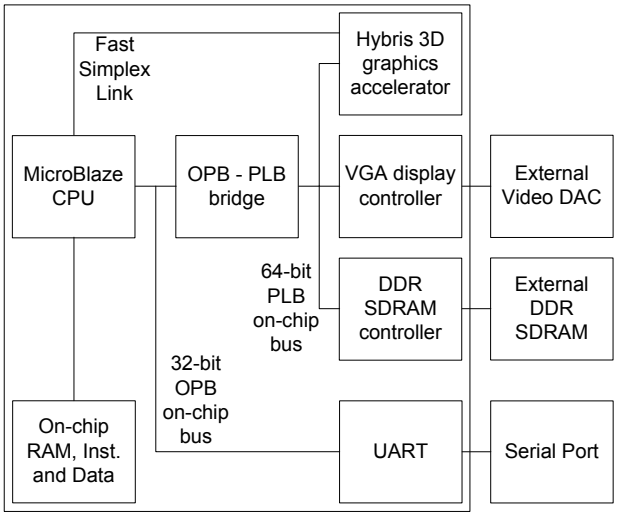

Figure 1. Overview of the embedded graphics system.

4. A SoC with CPU and Graphics Core

An overview of the System-on-Chip is presented in Figure 1. In addition to the graphics and memory subsystem the system contains the MicroBlaze CPU and two on-chip buses. We need to use the 64-bit PLB (Processor Local Bus) on-chip bus to support the performance requirements of the graphics system, while the 32-bit OPB (On-chip Peripheral Bus) is required for the MicroBlaze CPU. The on-chip buses are bridged via an OPB-to-PLB bus bridge which has a slave interface on the OPB side and a master interface on the PLB side. The bridge allows the CPU to initiate accesses to the devices on the PLB bus.

An important part of the PLB-side of the system is the main memory subsystem which is controlled by the DDR SDRAM controller. The DDR controller provides access to external DDR memory via the PLB-bus. The PLB bus bandwidth is dimensioned so it will match the bandwidth of the external memory. In practice this is accomplished by using 32-bit wide DDR SDRAM, which is represented internally as single data rate 64-bits on the PLB on-chip bus. If wider, e.g. 64-bit, DDR SDRAM is used the PLB on-chip bus may not be fast enough to match the memory bandwidth, as we cannot change the bus-width of the PLB, only the frequency. In such a case a dedicated link between the graphics core and the DDR memory controller core will be necessary. The DDR SDRAM in this system is built from two infineon 16Mbit x 16 chips which each

internally uses four 8192 row by 256x32 column DRAM banks, forming 64 MBytes of main memory with a page size of 4 kBytes.

The other devices on the PLB bus are a VGA video display core and the 3D graphics accelerator itself. The VGA video display core is a simple framebuffer display system reading pixels from a framebuffer in main memory. It stores an entire scanline of pixels in on-chip memory which allows it to use a fixed pixel clock for sending pixel data to the external video DAC, while periodically reading from the framebuffer at the full bus-speed. The VGA video display core is a PLB bus-master since it initiates the read transactions for reading from the framebuffer.

The framebuffer memory area in the main memory can be written to by the CPU (through the bus bridge), or by the 3D graphics accelerator core. This makes it possible to combine software rendering running on the CPU with hardware rendered pixels from the 3D graphics core.

The 3D graphics core needs to read from a list of triangles to render, and also needs to write to the framebuffer. To minimize the bandwidth requirement on the PLB on-chip bus and external memory, we use an on-chip scratch-pad memory in the 3D graphics core. The scratch-pad memory is used for rendering pixels in a single e.g. 32 x 32 pixel tile, we will later find the optimal tile size. By rendering the pixels into a scratch-pad RAM, we can later use a block-transfer to write a completed tile into the framebuffer. This similar to the virtual local framebuffer scheme as described in [5], which is often known as tile-based rendering. To support tile-based rendering we must first sort all triangles to be rendered into buckets corresponding to the tiles they overlap, see figure 2. Bucket sorting is an additional step not found in most commercial rendering hardware because of its added complexity and increased per-triangle cost and the requirement of a memory buffer for storing the bucket sorted triangles (figure 3). This imposes an upper limit on the number of triangles we can process in a single pass. However we gain several advantage with the on-chip scratch-pad memory which can be used e.g. for cheap depth-buffering using only on-chip memory. We can also use a simple solution for anti-aliasing where a supersampled image is first rendered to the scratch-pad memory and then down-filtered when writing the resulting tile to the framebuffer. This way a cheap way to do anti-aliasing can be provided for low-resolution embedded displays to improve the visual quality of the graphics. With minor changes to the rendering algorithms it is also possible to use sparse supersampling instead of normal full supersampling, Sparse supersampling focuses on improving the visual quality by focusing the sample rate for nearly vertical and horizontal edges, which is where the human eye is most sensitive to aliasing artifacts. This can be achieved by using one of the sub-pixel sampling patterns in figure 4.

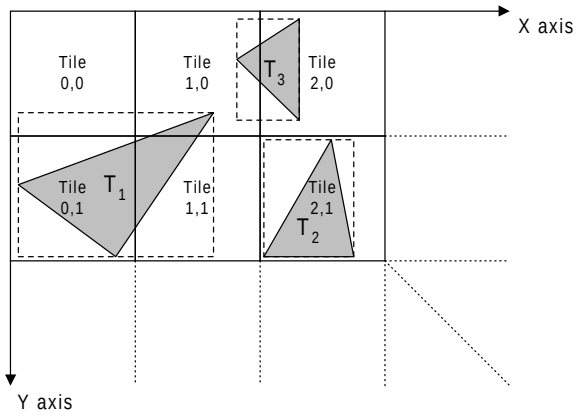

Figure 2. Examples of overlap in a tile-based renderer. Triangle T_2 is completely inside one tile. T_3 overlaps two tiles. T_1 overlaps 4 tiles if bounding box bucket sorting is used, but will only overlap 3 tiles if exact bucket sorting is used.

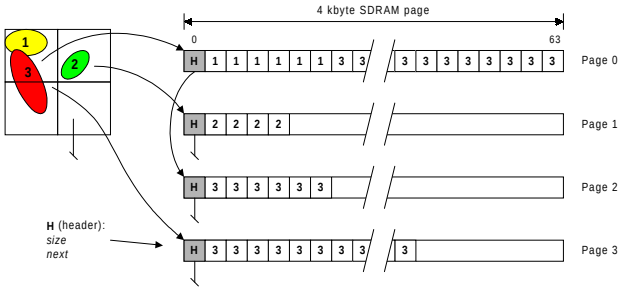

Figure 3. Memory management of triangle nodes in the bucket sorted triangle heap. Triangles are allocated in buffers of 63 triangles. A linked list of buffers are used, if space for more triangles are needed in a bucket. In this example the triangles of object 1 go to the first buffer, page 0. Object 2 is placed in page 1. The triangles of object 3 overlapping the upper left tile fills up page 0 and is continued in page 2. The triangles overlapping the lower left tile goes to page 3. The lower right tile is empty and nothing is allocated for it.

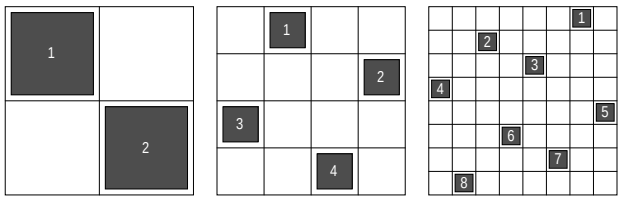

Figure 4. Sparse supersampling sub-pixel sample positions within a pixel. Left: 2 samples in a 2x2 grid. Middle: 4 samples in a 4x4 grid, Right: 8 samples in a 8x8 grid.

5. Optimizing the Hybris rendering engine

When building an embedded graphics system we must optimize the datapaths so their performance matches what is achievable given the limitations of the external main memory, internal on-chip scratch-pad memory, CPU caches and on-chip buses.

With a tile-based rendering engine one important parameter is the tile size. The size of a tile is mainly dictated by the available on-chip RAM resources, and also by the overlap factor which affects the overhead of the bucket sorting step. A smaller tile size reduces the demands of on-chip memory, but will also result in a larger overlap factor as the probability of a triangle overlapping multiple tiles will increase. A larger tile size will improve the memory coherence within the tile and will also help to reduce the overlap factor as a triangle will be less likely to overlap multiple tiles. Figure 2 shows a few cases of triangle-tile overlap.

This also depends on the triangle size distribution in the 3D scenes to be rendered. Assuming a typical triangle area of up to 64 pixels will keep the average overlap factor below 2 when using a tile size of 32x32 pixels. This is documented in [6].

Now that we know the bandwidth and memory limitations, we need to use this knowledge to properly configure the graphics architecture for a suitable implementation.

For the VGA video display we need to allocate main memory bandwidth for video refresh. With 640 x 480 32-bit pixels at 60 Hz and a 25 MHz pixel clock, the VGA display core will require an average bandwidth of 74 MBytes/sec from the main memory (100 MBytes/sec during active display and 0 during the blanking period). Updating the framebuffer at the same rate will require another 74 MBytes/sec.

The MicroBlaze CPU running at 100 MHz can at peak use up to 400 MBytes/sec bandwidth through the 32-bit OPB-to-PLB bus bridge. This is however not very likely, because the CPU is configured to use 16 kByte instruction- and data caches. The real bandwidth used by the CPU is difficult to predict, but is far below the peak figure mentioned before. If necessary, the CPU can also run (small) programs from internal block ram based memory.

This leaves a worst case main memory bandwidth of at least 800-400-100-74 = 226 MBytes/sec available for other use. We can use this for either a higher resolution framebuffer, or for the 3D graphics accelerator core.

The 3D graphics accelerator core will also need to read from a bucket sorted head of triangles from the main memory. The bandwidth required for this depends on the number of triangles in the scene, the size of a triangle description node, the triangle tile overlap factor and the frame rate.

Any remaining bandwidth can be used for other purposes such as a future implementation of texture mapping, or having a global depth (Z) buffer in main memory. Note that the tile-rendering algorithm does not require a global depth buffer. However if the number of triangles in a scene exceeds the memory allocated for the bucket sorted triangle heap, a multipass algorithm could be used to save the depth-buffer tiles to main-memory between the passes.

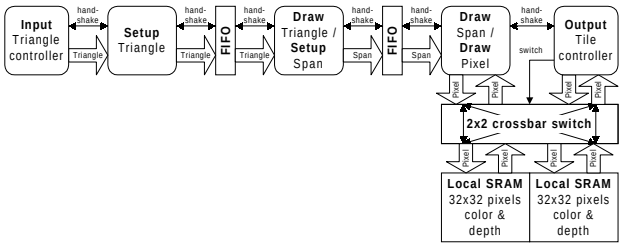

Figure 5. Architectural overview of the tile rendering engine back-end pipeline. FIFO buffers are placed between iterating pipeline stages to help average out load imbalances. The double buffered tile buffer allows the tile engine to render one tile while the previously rendered tile is being copied to the global framebuffer.

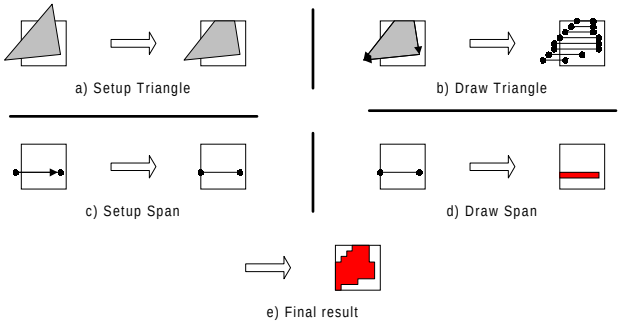

Figure 6. Processes in the tile rendering engine back-end pipeline. a) Setup Triangle adjusts y-parameters to fit tile. b) Draw Triangle iterates over the active scanlines, generating spans. c) Setup Span adjusts x-parameters to fit tile. d) Draw Span iterates over the active pixels in the span, performing per-pixel shading and depth testing. e) Final result for drawing one triangle.

5.1. Rendering core performance

The 3D graphics core in figure 5 was implemented as a core for the FPGA. Figure 6 shows how a triangle is rasterized by the tile rendering engine. The small FIFO buffers provide dynamic load balancing between the stages. The last pixel drawing stage is able to render a pixel on every clock cycle, provided that the previous stages can supply data fast enough. For this reason the previous stages should be designed so that the per-triangle cycle count matches the per-scanline (times the number of scanlines) cycle count and also the per-pixel cost (times the triangle area). If we only render relatively large triangles it

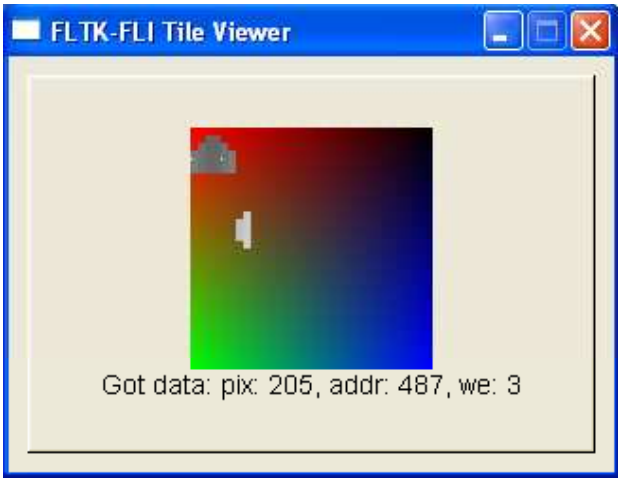

Figure 7. Graphical testbench window used for verification of the tile rendering engine. The testbench was written in C using FLTK for graphics and connects to ModelSim using FLI (Foreign Language Interface). Each pixel in the 32x32 tile has been magnified to a 4x4 pixel block and the shaded background makes it easier to see both dark and bright pixels.

is sufficient to have a fast pixel drawing stage. Some applications require a large number of small triangles (this is the trend today for increasing detail), one example is medical visualization where we usually end up with millions of triangles in a reconstructed 3D surface. For such applications, we anticipate a large number of small triangles which places significant demands on per-triangle processing as well. As a result our final implementation is dimensioned so it is able to handle very small triangles without slowing the pixel drawing rate, i.e. the per-triangle and per-scanline processing time is also one clock cycle, i.e. we have single cycle triangle setup and scanline setup. If the triangle area is larger than a few pixels the tile rendering engine can be balanced to improve the pixel throughput relative to the per-triangle throughput, this can be done by reducing the hardware resources in the triangle and scanline stages by using a multicycle machine for computing the initial setup. For performance reasons another approach is to speed up the pixel processing speed instead, which can be done by using parallel pixel pipelines [6].

The rendering core has been tested in simulation using ModelSim and also synthesized and tested on the FPGA itself. A 32x32 pixel tile engine capable of rendering one one pixel triangle per clock cycle occupies 2107 slices and 3 RAMB16 in a Virtex4 LX25, and can operate at 70 MHz with single cycle triangle processing. The critical path here is in the triangle setup stage, providing fast performance for small triangles. If faster per-triangle processing speed is needed we can pipeline the triangle setup and the scanline setup stages. The design is well suited for automatic pipelining by adding several registers to the output of each stage and then synthesizing it with register balancing turned on in a synthesis tool which supports it. Depending on the triangle size distribution in the application we can also allow the triangle setup to take multiple cycles, either using a FSM controller/datapath or use a simpler setup with a multicycle combinational path.

Note that the above results are for 8 bits per pixel + 32 bit per pixel for the Z (depth) buffer used in the tile rendering algorithm, which is mapped to the 3 block rams (single buffered, 6 are needed in the double buffered version shown in figure 5). Note that the tile size can be adjusted to match the number of block rams available for the tile rendering engine, the desired number of bits per pixel and whether we need single or double buffering. The tile size also affects the overlap factor during bucket sorting of the triangles prior to rendering; larger tiles give lower overlap, but use more memory and makes load balancing worse in a parallel implementation. In [6] we show that a tile size of ca. 32x32 pixels is a good compromise. This has also been shown recently in [1].

For verification we used a specialized graphical testbench written in C, which interfaces to ModelSim SE through its FLI interface. This graphical testbench provides a quick visual overview of what is written to the tile pixel- and depth-buffers cycle by cycle. In practice this is done by writing a VHDL entity which has the relevant signals as input ports which then connects to the compiled testbench which was written in C and uses FLTK to display a graphics window. Figure 7 shows an example of the graphical testbench window. The synthesized tile rendering engine has also been verified to work in hardware using a VGA monitor to display data being read directly from the tile buffer, showing the same images as the graphical testbench. Figure 8 shows an example of the type of graphics that can be rendered with the graphics core.

6. Summary and future work

The presented graphics system can be tuned for performance depending on the target application. As a starting point we are able to rasterize small triangles at a high rate. Rendering one one pixel triangle per cycle at 70 MHz gives a peak triangle processing rate of 70 million triangles per second. This assumes a memory system fast enough to supply data to the tile engine. Internally each triangle is sent to the tile rendering engine on a 333 bit wide bus in a single cycle. This translates to a peak bandwidth of roughly 3 Gbytes/sec for reading the input data. The total bandwidth is only 800 MBytes/sec on the FPGA evaluation board, so we cannot stream data fast enough from the main memory. One way to solve this data memory bandwidth problem could be to store a small dataset in the FPGA block rams, although that would limit the usefulness of the system.

Note that the above bandwidth is a worst case calculation, if the graphics system is used to render more typical

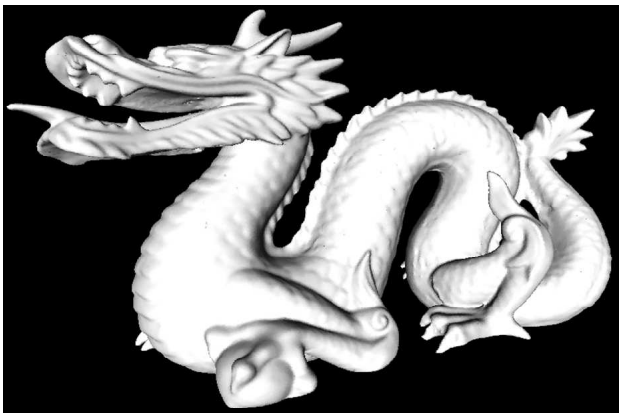

Figure 8. Example of an object rendered with the FPGA-based 3D graphics core. This is the Standford Dragon laserscan with 870000 triangles. The image was rendered in 2 seconds on the Virtex 1000 FPGA board running at 25 MHz.

scenes with larger triangles with an average area of about 64 pixels, the bandwidth requirement for the external triangle buffer will also drop by a factor 64, in the case of a tile rendering engine rendering one pixel per cycle. Triangles with an average area of 64 pixels also provide a good match for the 32x32 pixel tile size, giving an overlap factor of 2.

To fully utilize the speed potential in this graphics system, we need faster off-chip memory similar to the wide and fast DDR2 SDRAM used in modern graphics cards. Alternatively we can suggest that an evaluation board such as the ML401 is more suited for processing intensive rather than memory intensive applications. A Spartan 3E FPGA with wider and faster external RAM might be better suited for a low-cost implementation.

A complete System-on-Chip for an embedded graphics system will also need to do transformation, lighting and triangle setup before sending the triangles to the tile rendering engine for rasterization. While this can be handled in software on the MicroBlaze CPU version 4.0 with a hardware floating point unit, a dedicated transformation and lighting module can be implemented using e.g. the new Xilinx core-generator floating point cores. A dedicated vector processing unit will make it possible create a balanced graphics system that can both generate transformed and lit triangles, and also render them at the same rate. In the future is will be interesting to investigate the use of the PowerPC hard-core CPU found in the new Virtex-4 FX FPGAs, as it provides an Auxiliary Processor Unit interface to accelerator cores similar to the FSL links found in the MicroBlaze. However we must be careful not to build a system that requires an FPGA which is too expensive, which would make it difficult to compete against systems using dedicated ASICs.

References

[1] I. Antochi, B. Juurlink, and S. Vassiliadis. Selecting the optimal tile size for low-power tile-based rendering. In *Proceedings ProRISC 2002*, pages 1–6, November 2002.

[2] I. Antochi, B. Juurlink, S. Vassiliadis, and P. Liuha. Memory bandwidth requirements of tile-based rendering. In *Proceedings of the Third and Fourth International Workshops SAMOS 2003 and SAMOS 2004 (LNCS 3133)*, pages 323–332, July 2004.

[3] T.-C. Chiueh. Heresy: A virtual image-space 3d rasterization architecture. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 69–77, August 1997.

[4] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover. PixelFlow: The realization. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 57–68, August 1997.

[5] Foley, van Dam, Feiner, and Hughes. *Computer Graphics, Principles and Practice*. Addison-Wesley, second edition, 1990.

[6] H. Holten-Lund. *Design for Scalability in 3D Computer Graphics Architectures*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, July 2001.

[7] H. Holten-Lund. FPGA-based 3D graphics processor with PCI-bus interface, an implementation case study. *NORCHIP 2002 Proceedings*, pages 316–321, November 2002.

[8] M. Kelley, S. Winner, and K. Gould. A scalable hardware render accelerator using a modified scanline algorithm. *SIGGRAPH Proceedings*, pages 241–248, July 1992.

[9] S. Nishimura and T. L. Kunii. Vc-1: A scalable graphics computer with virtual local frame buffers. *SIGGRAPH Proceedings*, pages 365–372, August 1996.

[10] Xilinx. *MicroBlaze Processor Reference Guide, UG081*, v5.1 edition, April 2005.

RTL Structuring Using a Schematic Based Code Frame Generator for HDLs

Jörg Lastig
University of Applied Sciences Hamburg
Berliner Tor 7
20099 Hamburg, Germany
joerg@lastig.de

Jürgen Reichardt
University of Applied Sciences Hamburg
Berliner Tor 7
20099 Hamburg, Germany
reichardt@etech.haw-hamburg.de

Abstract

The paper describes design and application of a VHDL RTL code frame generator which is the first step in a more advanced project that shall support the RTL-design process with an always up-to-date ("live") block diagram of the system architecture. Our approach uses Microsoft Visio based schematic entry because it enables a cheap and flexible code generator implementation with large flexibility for different HDL's including SystemC. In order to demonstrate practical application of this tool we will show the VHDL design of a PCM31 framer core FPGA for telecommunication applications.

1. Introduction

The design of complex digital systems normally requires a large number of modules or components many of those having much more than one process. In fact on RT level all those processes represent functional blocks which can be synthesised to either synchronous or combinational hardware elements. While nowadays HDL's are widely used for the RTL-design of the components [5] it is very helpful for the design process to use a graphical representation of the block diagram which describes the partitioning into components and processes and their communication using signals.
A "live" documentation supports and eases the design process as follows:

- The initial design is captured graphically and the HDL code frame is generated automatically.
- Desired hardware functionality of processes is added to the code frame using a text editor. Eventually also architectural changes are made by adding / deleting components, processes or signals.
- Any code which changed the initial architecture requires a reverse operation which generates a block diagram from a HDL description.

The advantages of this approach are:

- A graphical description of the architecture is more comprehensive than a textual (HDL) description.
- The graphical description must be always consistent with the HDL code.
- Writing of HDL code is reduced to describing the real functionality. No redundant code writing is required. In VHDL for example all component declarations and -instantiations are added automatically.
- Consistent signal names and data types are forced.
- Errors due to semantic differences in simulation and synthesis are suppressed by the automatic generation of process sensitivity lists.

In this paper we describe the first step of this project, i.e. the code frame generator. In contrast to other commercial approaches we used MS Visio 2003 for schematic entry due to the following reasons:

- A graphical representation of the RTL-objects can be easily defined.

- The RTL-objects can carry attributes which describe their interfaces and behaviour.
- Object attributes which are embedded in Visio files can be easily processed to generate a netlist. This is supported by Visio Basic for Applications (VBA) programs.
- MS-Visio 2003 is widely used and requires no special knowledge if object modifications are required.

The netlist which has been generated by a VBA program can be converted to the HDL code frame by using a HDL-specific application program. Fig. 1 shows the workflow with the Visio2003 based code frame generator.

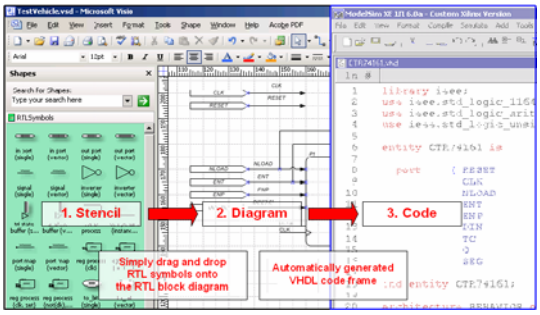

Figure 1. Workflow with the HDL code frame generator: Symbols are taken from the MS-Visio HDL-stencil library (1). They are placed and connected on a diagram sheet (2). Finally the code frame is being generated automatically (3).

2. Tools in the design process

A ShapeSheet is a textual spreadsheet behind the graphical view of every Visio drawing [3]. This sheet contains all drawing information that each shape comprises. The ShapeSheet consists of sections and each section includes multiple rows. Each row of every section can contain formulas or absolute values just like the cells of a spreadsheet. ShapeSheet programming is used for the realisation of the context menus for most of the RTL symbols[4]. The hardware designer can select and modify one or more attributes while creating his RTL block diagram.
These features of Visio 2003 enable the initial design flow which is shown in Fig.2.:
Partitioning of the hardware structure down to individual processes is not done with a HDL editor but graphically. A HDL code frame is generated automatically. As also shown in Fig. 2. the functionality of the digital hardware is implemented later using a HDL text editor.
If the design process requires architecture modifications they are most probably done by a modification of the HDL code. So there is demand for additional tools which perform HDL-code parsing, graphical object generation and intelligent placement and wiring of graphical objects (see Fig. 3).

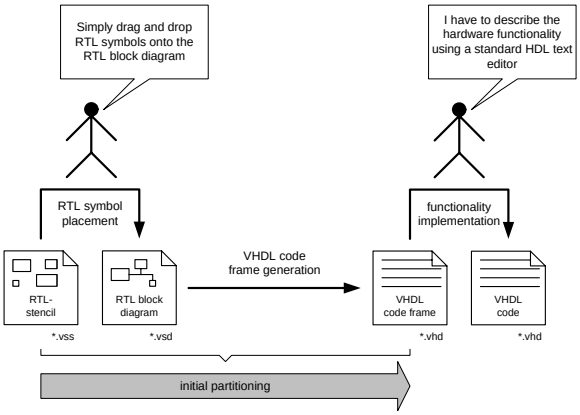

Figure 2. The initial partitioning of architectural elements uses the code frame generator on the basis of Visio 2003.

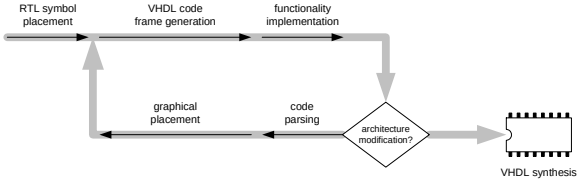

Figure 3. Architecture modifications in the HDL code require additional code parsing and graphical placement tools.

In our approach a complete RTL design can be described in a single Visio 2003 design file (*.vsd). In this design each drawing page corresponds to an individual module / entity. The RTL block diagram is generated by dragging shapes from an RTL-stencil library (*.vss) onto the drawing page. The stencil contains a set of MasterShapes that can be simply dragged and dropped into the RTL block diagram. Those shapes are called SmartShapes because for example they can

carry information where to connect with each other [3]. They may have different data types and contain other smarts. All SmartShapes which are defined in the RTL-library consist of simplified graphical elements to guarantee easy recognition. Their attributes are programmed with user friendly behavior which eases the RTL design process.
The netlist module which is shown in Fig. 2 uses the automation interface of the Microsoft Visio object model. This module represents the objects and methods that the Visio engine initialises via automation and describes how the objects are related to each other. The Microsoft Visual Basic for Applications (VBA) code of the netlist module runs within an instance of Visio and accesses the objects it requires. The netlist module analyses the entire block diagram of an entity. This analysis includes mapping of connected signals and components into a textual netlist file (*.txt). The mapping embeds also the attributes of the ShapeSheet.
The generic netlist is used as an internal interface to transfer connection information between the netlist generator module and the VHDL code frame generator module. The structure of the netlist allows easy extension of the code frame generator to other HDL languages.

3. The code frame generator

Although the concept allows support of more advanced HDLs like SystemC the actual version only generates VHDL code. The VHDL code frame generator module is implemented as a Win32 console application in object oriented C++ [8]. It imports the netlist information via program arguments and generates the VHDL code frame as output. Each RTL symbol group belongs to C++ methods in agreement with the VHDL syntax generation. The code frame generator uses standard C/C++ libraries only in order to be flexible with future requirements and to be able to implement the code with different C/C++ compilers.

The VHDL code frame generator module will be invoked for individual components after simply selecting the corresponding RTL-sheet in the design and pressing a predefined hotkey in Visio. The VHDL file will be named automatically with the entity name of the sheet. Presently the VHDL generator produces a code frame with the following information:

- Entity declaration with I/O port information and inheritance of port-names to internal signals.
- Local signal and bus declarations with data type assignment.
- Component declarations and instantiations with port map assignments.
- Clocked processes with appropriate sensitivity list.
- Combinational processes with appropriate sensitivity list.
- Concurrent VHDL signal assignments.
- Automatic wiring of a predefined clock- and reset signal within a single component.

4. The demonstrator: A PCM31 framer core

A Framer is a device used to align or synchronise to an embedded framing pattern in a serial bit stream. Once synchronised and data fields are properly aligned, overhead bits for alarms, performance monitoring or embedded signalling may be extracted and processed. The designed PCM31 framer core is used as a demonstrator for the code frame approach. It implements ITU recommendations G.703 [1] and G.704 [2]: A flexible framer core can be used for multiplexing a PCM stream at 2048 kHz for n x 64 Kbps applications where n is a scaling parameter which is set according to the user defined application (see Fig. 4).

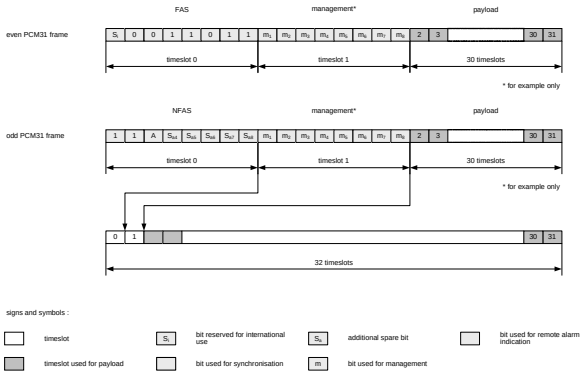

Figure 4. Description of the PCM31 standard according to G.704 [2]

From Fig 5 it can be seen that the FPGA implementation of the PCM31 core uses three clock domains. They use a common DPLL to provide a synchronous transmit clock (TX clock). The external system clock must be 32.768 MHz to ensure proper operation of the DPLL. Internal FIFO interfaces run at different speed (with independent read and write clocks) and they are used to transfer data between the three different clock domains.

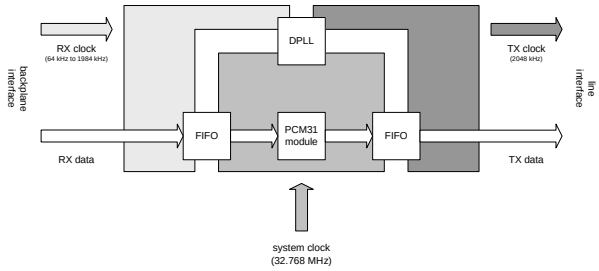

Figure 5. Clock domains of the PCM31 framer.

Fig. 6. shows the top-level architecture of the design as defined by the RTL-stencil library. It specifically shows the component instantiation of a 3-State Phase Frequency Detector (PFD) in the RTL block diagram of the DPLL. The task of the PFD is the detection of differences in phase and frequency between the reference and DCO (Digital Controlled Oscillator) clock. The output of this component / module are the PFD_UP and PFD_DN signals. Fig. 6 also shows a screenshot of the automatically generated VHDL code frame in the background.

5. Conclusions

As the first step towards a "live" graphical documentation of RTL-architectures we have developed a code frame generator using Visio 2003. This approach has been chosen in order to support a cheap solution which is easy to use, has simple interfaces and can be easily modified. We have shown that the use of the code frame generator is less error prone than a purely HDL approach because it not only relieves the designer from writing numerous redundant VHDL code lines but also forces the designers to use consistent names and data types. Finally the design was inherently free from any errors due to different simulation and synthesis semantics because the sensitivity list of processes were created automatically in agreement with the VHDL RTL synthesis standard [13].

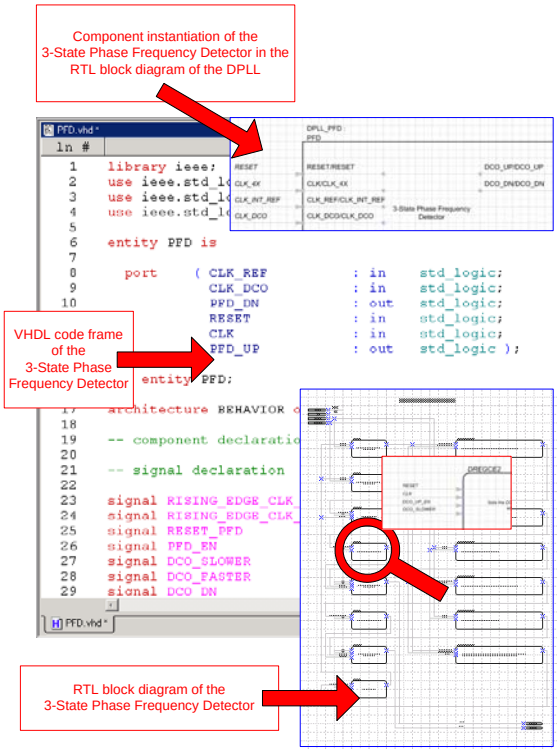

Figure 6. Top level architecture diagram of the PCM31 framer as drawn in Visio 2003 (lower right). Component instantiation of the Phase Frequency Detector (upper right) and automatically generated VHDL code frame which shows the PFD entity definition (background).

The underlying concept can be easily extended to more advanced HDLs like SystemC which have large similarities with VHDL on RT level [11] [12].
The promising results of this first project milestone gives large motivation for the second part of the project which we will start in the near future: I.e. the design of a program which reads and interprets the VHDL code and automatically generates corresponding ShapeSheets for Visio 2003.

References

[1] G.703 (11/01) Physical / electrical characteristics of hierarchical digital interfaces; http://www.itu.int

[2] G.704 (10/98) Synchronous frame structures used at 1544, 6312, 2048, 8448 and 44 736 kbit/s hierarchical levels; http://www.itu.int

[3] Bonnie Biafore; Visio 2003 Bible; 2003; Wiley Publishing

[4] Senaj Lelic, Helmut Schuster, Matthias Reich; 2002; Das Microsoft Visio 2002 Handbuch; 2002; Microsoft Press

[5] Jürgen Reichardt, Bernd Schwarz; 2003; VHDL Synthese; Oldenbourg

[6] Peter J. Ashenden; 2002; The Designers Guide to VHDL; Publishers Morgan Kaufmann

[7] Ulla Kirch Prinz, Peter Prinz; 2003 C++ für C-Programmierer; Galileo Press

[8] Ulla Kirch Prinz, Peter Prinz; 2001; Alles zur objektorientierten Programmierung in C++; Galileo Press

[9] Rene Martin; 2002; Visio 2002 für Anwender; Software and Support

[10] Mark H. Walker, Nanette Eaton; 2003; Microsoft Office Visio 2003 Inside Out; Microsoft Press

[11] J. Bhasker,; A SystemC Primer, Star Galaxy Publishing

[12] http://www.systemc.org

[13] 1076.6-1999, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, Institute of Electrical and Electronics Engineers Inc., 1999, ISBN 0-7381-1819-2

A Soft Fixed-Point Digital Signal Processor Applied in Power Electronics

S. de Pablo, J. Cebrián, L. C. Herrero
University of Valladolid
E.T.S.I.I., Paseo del Cauce, s/n
47011 Valladolid (Spain)
sanpab@eis.uva.es

A. B. Rey
Polytechnic University of Cartagena
Edif. Antiguo Hospital de Marina
30202 Cartagena, Murcia (Spain)
alexis.rey@upct.es

Abstract

*This article presents the "DSPuva16" processor, specifically developed for Power Electronics applications. As its name indicates, this is a Digital Signal Processor oriented to calculation: it operates using fixed-point 16-bit numbers extending its precision up to 24 bits. The MAC operations (rD = rD ± rS * rT) are made regularly using only one instruction cycle, that requires four clock cycles. The processor has been physically tested on FPGA working at 40 MHz.*

The application of this processor in Power Electronics (distributed generation systems, AC motor control, active filters) is quite natural, because all working magnitudes are limited in range. The use of several processors is also possible: each one would execute a different regulation loop, with working cycles between 5 and 100 microseconds.

1. Introduction

The first devices used to control electronic power converters (rectifiers, inverters) were analog circuits based on op-amps [7]. When first microprocessors appeared, they were immediately implemented in the control circuits [5] replacing their equivalent analog ones because of their inherent advantages of stability, noise immunity and facility of adjustment. Later we saw the rise of digital signal processors, with greater calculation capacity, which allowed a more precise control and to reach remarkably superior performances [2].

At the moment we are in a point in which, clearly, a processor alone does not suffices for controlling all the system: lots of regulation loops must be controlled, some of them working at high frequency (near the microsecond) [6]. In addition, other tasks such as remote communication or user interface must be attended with increasing behavior demands.

The obvious solution is to use several processors to take care of these diverse tasks. It is not only cheaper and easier to control them that way instead of using only one high performance processor, but also that is the only way to guarantee control in real time.

What happens if the possibility of connecting several processors in a printed circuit board is evaluated? Immediately we get involved in new problems: a high number of nets are required for the high data flow between processors, because the standard serial channels are not usually enough. So additional intermediate elements must be added to accommodate the different data flows. Besides, in general, all used components quickly become obsolete, which forces to redesign all systems once and again. All of these topics lead inevitably to elevated NRE costs.

The panorama is completely different if soft processors are used. Possibly their performance is not as good as their counterpart hard ones, but when they become all integrated in a chip (FPGA or ASIC) all the previous problems disappear immediately. Communication between processors is flexible and immediate using dual-port synchronous memories [4], whose use absolutely does not affect the cost of the equipment [1]. Obsolescence is null since previous designs can be synthesized again, targeting new devices, which will be able to work at higher frequencies.

The main limitation we have, specially in the FPGA domain, is the impossibility of using floating-point units, because their use is absolutely prohibitive if they are not hard-wired on the chip. Nevertheless, it is also possible and it is not excessively complex the use of fixed-point arithmetic. All magnitudes (voltages, currents, gains of regulators, etc.) are naturally or artificially limited by working restrictions [6].

Therefore, in this applications field it would be normal the use of several fixed-point digital signal processors and one or two general purpose processors to take care of communications and user attention. Their communication inside FPGA or ASIC devices is not expensive: there are no restrictions in the amount of communication lines, because they are all internal. The bandwidth is not a problem either, because dual-port memories may work at system frequency and use very low space.

This article presents a fixed-point DSP with enough performance and small size, prepared to handle a reduced set of variables. It has been designed thinking that it could be better to use several small and connected processors solving different loosely coupled tasks, than a huge processor doing all the work. The latter would force designers to use a considerably greater clock frequency and a complicated interruption scheme to attend all the real time demands. On the other hand, several processors may execute different regulation loops employing different working periods, but they are easily coupled as seen on section 7 and demonstrated in [1].

2. Main features of the DSPuva16

The DSPuva16 is a 16-bit fixed-point digital signal processor that extends its precision up to 24 bits. It uses what is called Harvard architecture, because it gets instructions using dedicated buses to its program memory (whose size varies between 256x16 and 4Kx16, according to the processor model) and exchanges data (up to 256 synchronous 16-bits ports) using other buses.

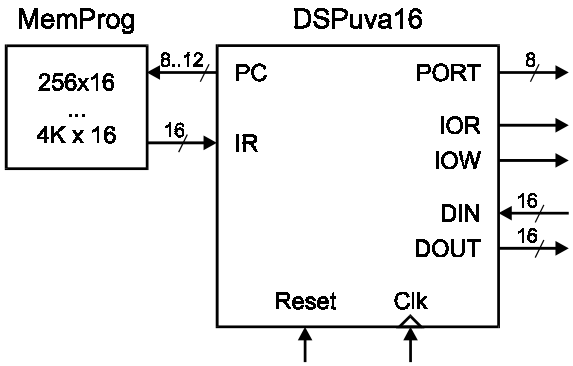

Figure 1. External connections of the DSP.

Its basic operation is the multiplication with accumulation, either positive or negative (rD = rD ± rS * rT), that it regularly executes in an instruction cycle. This one is always made using four clock cycles, as it corresponds to its RISC architecture.

It has 16 registers of 24 bits, denominated from 'r0' to 'r15', that can be used in any operation except for 'r0', which cannot be used as operand. Indeed this characteristic allows us to considerably extend the possibilities of the processor without adding excessive complexity to the internal design. When the 'r0' codification is found in the location of 'rS', its value is immediately annulled, and when it is found in the position of 'rT', its value is replaced by a 16-bit constant that is taken from the program memory. No additional delay is introduced when a constant is called.

This scheme extends, for example, a basic processor instruction as it is the sum (rD = rS + rT). It allows operating with an immediate constant (rD = rS + K) and making direct assignments to any register (rD = 0 + rT; rD = 0 + K). This property is widely used in the programming of digital filters, whose general structure is:

$$y(t) = c_0 \cdot x(t) + c_1 \cdot x(t-T) + \ldots + d_1 \cdot y(t-T) + d_2 \cdot y(t-2T) + \ldots$$

where all coefficients are constant.

External accesses are made through 256 synchronous ports (each access is completed in only one clock cycle) with direct addressing: pN = rS; rD = pM; where N and M are any value between 0 and 255. It could be thought that this addressing mode is less flexible than the indirect one, but in the application field of this processor this does not mean any limitation: it is usual that ports correspond with memory positions used to extend the processor spartan storage capacity, or they may connect with physical devices that allow reading or generating analog measures. Anyway, indirect addressing is always possible using an external pointer, integrated in the same chip; we must remember that we are designing within an FPGA or ASIC.

The control of subroutines is similar to other processors, but the user must dedicate a register (usually 'r0') to keep safe the returning address; later on, the same register can be used to return to the calling point. If nested subroutines are desired, either another register or an external LIFO stack should be used.

3. The instruction set

The instruction set of the DSPuva16, as shown in the table 1, is very simple. It consists only of 17 different instructions, but it reaches a remarkable flexibility thanks to how it uses the codification corresponding to 'r0', as it has been explained in the previous section.

Table 1. DSPuva16 instruction set.

Opcode	Mnemonic	Operation
0000 aaaa aaaa dddd	call (rD) addr	Absolute jump
0001 0xxx xxxx ssss	ret (rS)	Return (pc = rS)
0001 1fff aaaa aaaa	jpFlag addr	Relative jump
0010 dddd nnnn nnnn	rD = pN	Read from port
0011 ssss nnnn nnnn	pN = rS	Write to port
0100 dddd ssss tttt	rD = rS * rT	Normal product
0101 dddd ssss tttt	rD = rS x rT	Shifter product
0110 dddd ssss tttt	rD = rD + rS * rT	Positive MAC
0111 dddd ssss tttt	rD = rD – rS * rT	Negative MAC
1000 xfff dddd tttt	ifFlag rD = rT	Cond. assignment
1001 xfff dddd tttt	ifFlag rD = –rT	Cond. assignment
1010 dddd ssss tttt	rD = rS + rT	Addition
1011 dddd ssss tttt	rD = rS – rT	Subtraction

1100 dddd ssss tttt	rD = rS and rT	Logic AND
1101 dddd ssss tttt	rD = rS or rT	Logic OR
1110 dddd ssss tttt	rD = rS nor rT	Logic NOR
1111 dddd ssss tttt	rD = rS xor rT	Logic XOR

Whenever a subroutine is called the returning address is kept in a register, usually 'r0'. The instruction code dedicates only eight bits to the destination address, so it seems that program length must be limited to only 256 instructions. This is more than enough for many cases, because it is preferred to use several processors executing different regulation loop, so programs are usually short. Anyway, the available program length has been extended using a simple and powerful mechanism: there are up to five different models for this processor (named 'A', 'B', 'C', 'D' and 'E'), with programs of up to 256, 512, 1K, 2K and 4K instructions. When an absolute jump takes place (with the instruction 'call'), it is only possible to jump to even positions when the 'B' model is used, only to one of each four if the model is 'C', and so on. That means that all subroutines must be correctly aligned in the program memory, using the "#align" assembler directive. The processor model may be specified using the "#model" directive.

Conditional jumps are always relative, just to avoid the alignment problem. Like many other processors, these jumps can be made to near positions (in a range of ±128 instructions, more or less), and the typical conditions are the usual ones: 'eq', 'ne', 'gt', 'ge', 'lt', 'le', 'v' and 'nv'. The same conditions can be used to make conditional allocations, as happens in "rN = rN; iflt rN = –rN", that calculates the absolute value of a number kept in a register.

This processor only use the direct addressing mode to read (rD = pN) and to write (pN = rS) on external ports. Indirect addressing is available using external pointers but they are not usually needed in power electronics applications, where only a reduced set of variables is managed and algorithms are executed regularly over the same data set. Physical accesses are synchronous and they are executed in only one clock cycle. This does not mean any limitation either, because external resources are implemented with the same technology, in the same chip, and therefore they can work at the same clock frequency.

Instructions that make products, additions, subtractions and logic operations, have a regular structure that address two operands (any register except 'r0') and a destination register (any one). As said before, the possibilities of the instruction set have been extended allowing the cancellation of the first operand ('rS') and the substitution of the second ('rT') by a 16-bit constant ('K'). In this way, immediate logic masks can be applied to (rD = rS and/or K), the content of a register can be inverted (rD = 0 nor rT) and any register can be initialized with a constant (rD = 0 + K).

The most important operation of this processor, and its reason of being, is the fixed-point product, with or without accumulation. In general, two 16-bit normalized[1] operands are taken and a 24-bit normalized result is produced. All accumulations, additions and subtractions are made with a resolution of 24 bits.

Another kind of product is also available, represented as "rD = rS x rT", which allows to make displacements. The second operand is interpreted in <8.8> format, so that any value can be multiplied by 1/128, 1/64..., 1/4, 1/2, 2, 4..., 32, 64, in addition to other intermediate values, resulting in the desired adjustment.

Other typical instructions have been implemented with "macros" recognized by the assembly language. The 'nop' instruction, that does nothing, is replaced by "r1 = r1 or r1" and the newly created 'break' instruction, which allows setting simulator break-points, is replaced by "r1 = r1 and r1", doing nothing too.

4. The instruction cycle

As indicated above, the internal architecture of this processor is RISC like. That means that it executes all of its instructions regularly, in four clock cycles particularly. Although, from the user's point of view, all instructions are executed in four cycles, certain overlapping between instructions actually exists, and in fact many instructions are finished when already the following one is being executed. In any case, only one latency effect must be considered, and it will be explained later.

All instructions begin reading their 16-bit operation code from the program memory, dedicating to this function two clock cycles. After that, they use other two cycles to read the operands, 'rS' first and then 'rT'. The latter is replaced by a constant read from the program memory if 'r0' is referenced. Finally, in the first clock cycle of the next instruction, the required operation is made and the result is stored in 'rD'.

1) It sends the PC to the program memory.
2) It gathers the operation code on IR.
3) The rS register is read to an 'ACC' register.
4) 'RegT' and 'RegS' get the rT and ACC values.
1') The processor executes rD = RegS op RegT.

The program counter 'PC' is increased during the phase '2' and, if a constant is read from the memory, also during the phase '4'. If the operation requires a jump ('call', 'ret', 'jpFlag'), the program counter is modified during the phase '4'.

Using this simple scheme all instruction except products can be executed. As we'll see in 5.2 section, the internal multiplier of this processor requires four clock

[1] Normalized values are always in the [-1,+1) range. When 16 bits are used to represent these values the highest value can be 0.99996948 and its format is named <1.15>: one integer bit, which contains the sign, and fifteen fractional bits.

cycles to complete its operation, thus requiring a much smaller size[2] and it does not deteriorate the processor working frequency. In order to carry out the products, with or without accumulation, four clock cycles of the next instruction and another two ones from the following one are used:

1) It sends the PC to the program memory.
2) It gathers the operation code on IR.
3) The rS register is read to an 'ACC' register.
4) 'RegT' and 'RegS' get the rT and ACC values.
1') First stage of the MAC using RegS and RegT.
2') Second stage of the MAC using RegS and RegT.
3') Third stage of the MAC using RegS and RegT.
4') Fourth stage of the MAC using RegS and RegT.
1") One cycle because of multiplier segmentation.
2") Reads 'rD' and accumulates the MAC result.

In this way a *latency* is introduced and programmers ought to consider it: the result of a product is not available for general use in the immediately following instruction, but in the later one. However, when it is used for accumulation the latency is avoided, because that operation is carried out a cycle later too. This issue can be understand with the following example:

```
r1 = 0.27          // A value is allocated on r1
r2 = r1 + 0.32     // Right done, because r1 is available
r3 = r1 * r2       // Correct product between r1 and r2
r3 = r3 + r2 * r2// This use of r3 is correct
nop                // We must wait for the result of r3
p4 = r3        // Now, but not before, r3 is r1*r2+r2*r2
```

Therefore, when the result of a product must be used, except when it is made to accumulate on a register, it is necessary to add a 'nop' or another instruction after the multiplication, in order to give time to the result to be calculated. This one is the only latency that introduces the segmented architecture of this processor.

5. Internal architecture

The DSPuva16 has been designed using Verilog and its based, as schematically shown in figure 2, on a RISC architecture that uses two 24-bit buses, one for operands and another one for results.

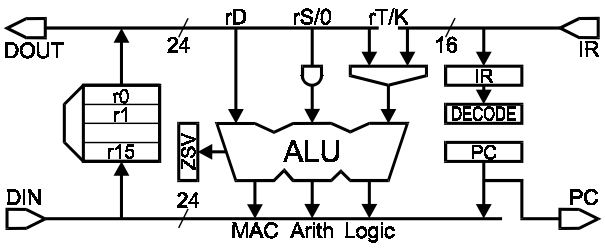

Figure 2. Internal architecture of the DSPuva16.

The program counter ('PC') has 8~12 bits, depending on the chosen processor model. The operation codes are received through 'IR' and they always have 16 bits. Using the registered value of IR a segmented instruction decoder activates each part of the circuit until completing each operation.

The bank of registers, that maintain the values of 'r0' to 'r15', is a 16x24 memory with synchronous writings and asynchronous reading [4]. It requires only 24 basic cells in some FPGA devices, which is equivalent to 5% of all the processor.

The ALU is built using three different units that realize logic operations, arithmetic ones and products. After each instruction, the zero ('Z'), sign ('S') and overflow ('V') flags are updated, which allows carrying out the corresponding allocations and conditional jumps. It must be pointed out that this processor does not need the carry flag ('C'), because it is not necessary for single precision operations.

Two different buses are used to interchange data with external elements, one to send data and another one to receive them. This method avoids additional tri-state buffers that, indeed, are not necessary within a chip.

5.1. Processor based on two internal buses

Many RISC processors have three or more buses at the moment, which even allow to realize all the instruction operations in only one clock cycle.

Since this processor needs four clock cycles to complete each product, and that the use of a third bus would lead to use dual-port memories for the bank of registers[3], it has been chosen to dedicate only one bus for the operands. Through this bus it can be seen the values of 'rD' during phases '1' and '2' and then it is left for 'rS' and 'rT' during the last two phases of each instruction cycle.

The bus for results is dedicated to collect the output of all ALUs and other sources: the input of external data when reading from a port and the value of the program counter when a subroutine is called, to keep the returning address in a register.

5.2. Four step multiplier

The central operation of this processor is the multiplication. It operates on two 16-bit values and generates a 32-bit result. Only 24 bits are available finally. In general the operands are in <1.15> format and the result is in <1.23> format.

In order to build the multiplier it has been decided to divide the operation in four stages[4], multiplying in each step a 16-bit operand by another one of only four bits,

[2] As the calculation of products is divided in four stages the circuit can be reduced to a quarter, approximately. Even so, the multiplier needs about 250 basic cells, that is a half of the processor.

[3] These memories allow two simultaneous asynchronous readings and one synchronous write when the clock cycle is finishing, but they occupy twice the space of the memories used by this processor.

[4] Not all FPGA devices do have embedded multipliers.

emitting a 20-bit intermediate result. This operation can be made with only four adders and an intermediate segmentation register, as shown simplified in figure 3. If we had tried to operate in a single clock cycle we would have needed 15 adders and the latency would have been greater, because of the segmentation registers.

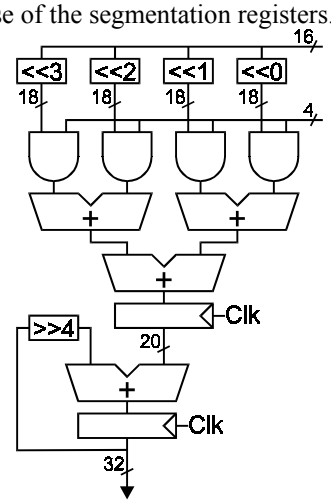

Figure 3. Segmented structure of the multiplier.

6. Development environment

The integrated development environment (IDE) of this processor fully covers its needs, at least while it is applied to the control of power electronics converters. Typical programs executed by the DSPuva16 processor usually have between 50 and 1.000 instructions, which correspond with typical working cycles of 5 to 100 microsecond. Therefore, programming in assembly language is sufficient, and the syntax used by the instructions is quite comfortable, as it has possibly been appreciated above.

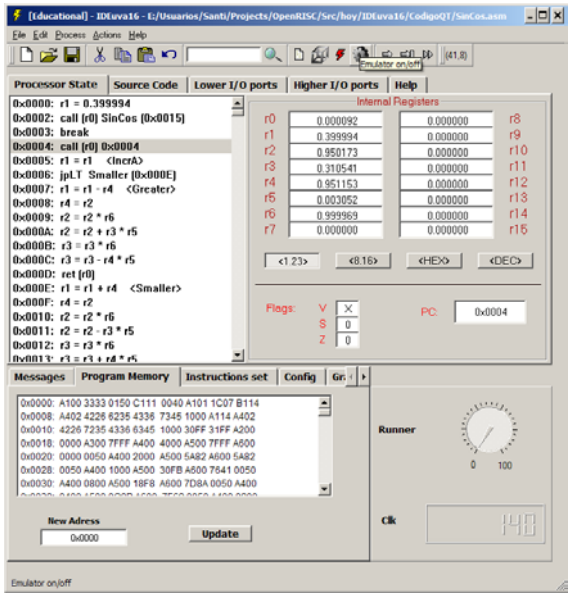

Figure 4. The IDE for the DSPuva16.

The IDEuva16 program is executed in graphics mode[5] and it incorporates a simple text editor, a complete C-like assembler with preprocessor, a simulator and a connection to the physical processor to control it in emulator mode using a suitable interface. Figure 4 displays the result of a small simulation in which the processor calculates "r2 = sin(r1); r3 = cos(r1)" for "r1 = 0.4·p". It needs 14.0 microseconds to complete this operation using a Cordic technique and working at 40 MHz (10 MIPS).

After editing and saving the assembler source code, where directives such as "#include", "#define" and "#ifdef" can be used, the source may be assembled by pressing a button. The process takes few seconds. Then, a step-by-step simulation or a simulation until a 'break' instruction can be done. Thousands or even hundreds of thousands of instructions can be executed in a single step, because simulation times are usually several or tens of milliseconds. The completion of each process on a typical DC/AC application takes few minutes using a 3 GHz computer. During simulations the state of the processor can be seen and intermediate results can be graphically displayed in a dynamic window.

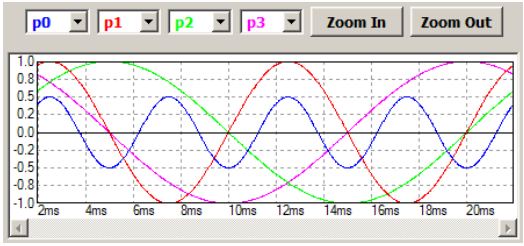

Figure 5. Graphics output of simulations.

When algorithms and assembler codes are stable enough, designers can use the "emulator mode": a real DSPuva16 is synthesized and implemented on an FPGA that is connected to the computer through the parallel port in SPP or EPP mode. The own IDE transfers to the DSP's program memory the result of the assembly, and following its state of reset/run is controlled. When the user stops the processor, the IDE automatically captures the intermediate or final results emitted by the processor through several of its ports, displaying them on the screen of the computer, as can be seen on figure 6.

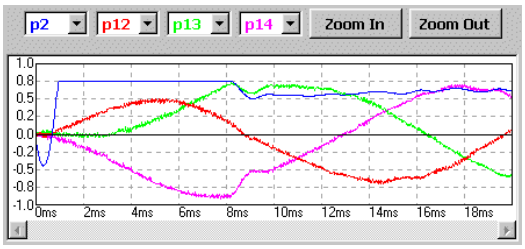

Figure 6. Graphics output of emulations.

[5] Currently English and Spanish versions on Win95/98/2K/XP are available. A Linux version will be released very soon.

The main difference between the simulation and the emulation is that the former is made completely in the computer, whereas the latter is executed in real or almost real time[6] on the physical equipment, implemented at the moment in a Xilinx Spartan-II FPGA[7].

7. Multi-processor systems

The main advantage of this DSP is its reduced size: it only needs about 12% of a 200K-gates FPGA. That means that, although a DSP alone cannot make the entire control task, is not too expensive to add others until the desired performance is reached. In a photovoltaic power application that is under development, and where this DSP is used for control, it is necessary to synchronize the equipment with the mains and to regulate the active and reactive powers; this tasks are covered by a first DSP with a working cycle of 25 microseconds. It is also necessary to control every 5 microseconds the DC/AC output currents and the commutation frequency of the equipment; for this task we have synthesized a second DSP. At the moment, the behaviour of the electrical circuit has been emulated every microsecond by a third DSP that operates in 60 microseconds. All this digital circuit requires about 100K gates (50%) of the FPGA.

For the communication between processors in this applications we use dual port synchronous memories [4]: a DSP can read and write at any time using one port, while another DSP can read, but not write, using the other one. For them, all accesses are as we have seen: direct accesses to ports. Using these memories, which occupy very little space in an FPGA, it is not necessary to worry about conflicts when sharing resources: both processors can use the common memory at any time. It is not necessary to synchronize transfers or signalling them either, because the bandwidth of involved magnitudes is far below the working cycle of these processors, reason why it does not matter that read values by the destination processor correspond with present values or previous ones, emitted by another DSP.

8. Conclusions

This article has shown how the DSPuva16 processor is and how, in spite of its limited features, it can easily solve complex problems of control in real time, simply adding as many processors as needed, but always within a chip, usually a FPGA.

Its main limitation is that it operates with few data and using fixed-point values, but since we have seen these topics do not mean any problem in the control of power electronic converters, because all physical magnitudes are easy to normalize and it is not difficult to keep them in a safe range, except during failure situations that cause the shutdown of the equipment and the stop of control.

9. Acknowledgments

Our thanks to Carmen Cascón [3] and Juan del Barrio [1], who work hard to develop the IDE of this processor and contribute to the first application of this DSP in the control of a photovoltaic power generation system. Thanks also to Le Duc Hung, from the University of Natural Sciences of Ho Chi Minh City (Vietnam) for his comments and improvements on this processor.

References

[1] J. del Barrio, *Desarrollo sobre FPGA de un Emulador de una Planta de Microgeneración Eléctrica*, Final Project at the ETSII, University of Valladolid, Spain, 2004.

[2] B. K. Bose, *Power Electronics and AC drives*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[3] C. Cascón, *Diseño de un entorno de desarrollo para un DSP en coma fija de 16/24 bits integrado en FPGA*, Final Project at the ETSII, University of Valladolid, Spain, 2003.

[4] S. K. Knapp, "XC4000 Series Edge-Triggered and Dual-Port RAM Capability", Xilinx XAPP065, 1996.

[5] B. Norris, *Electronic Power Control and Digital Techniques* (Texas Instruments Electronics Series), McGraw-Hill, 1976.

[6] A. B. Rey, *Control digital vectorial con sliding en fuente de corriente para convertidores CC/CA trifásicos conectados a red*, Ph.D. Thesis, University of Valladolid, Spain, 2000.

[7] J. Schaefer, *Rectifier Circuits: Theory and Design*, John Wiley & Sons, Inc., Library of Congress Catalog Card Number 65-12703, 1965.

---

6 Before beginning with the power tests on the real converter its better to physically prove the control of DSPs using an emulated electrical plant. This task can be carried out using another DSP, which calculates how the electrical circuit would behave. Nevertheless, its work cycle is usually much greater than the others, which forces to add additional delays at the others so that they must wait, loosing the pure real time characteristics. Anyway, FPGA emulation finishes in seconds what PC simulation takes several minutes [1].

7 The results of this article have been proven on a Xilinx XC2S200-PQ208 working at 40 MHz.

Call for Academic / Industrial Papers, Exhibits and Tutorials for FPGAworld September 2006, Stockholm, Sweden

The FPGAworld Conference addresses all aspects of digital and hardware/software system engineering on FPGA technology. It is a discussion and network forum for researchers and engineers working on industrial and research projects, state-of-the-art investigations, development, and applications.

The vision is to be the leading industrial and academic conference in FPGA technology.

Submission of Presentations
The submissions should be in at least one of these sessions:

- DESIGN METHODS - models and practices
  - Project methodology.
  - Design methods as Hardware/software co-design.
  - Modeling of different abstraction.
  - IP component designs.
  - Interface design: supporting modularity.
  - Integration - models and practices.
  - Verification and validation.
  - Board layout and verification.
  - Etc.
- TOOLS
  - News
  - Design, modeling, implementation, verification and validation.
  - Instrumentation, monitoring, testing, debugging etc.
  - Synthesis, Compilers and languages.
  - Etc.
- ARCHITECTURES
  - Platforms design, real-time operating systems, communication etc.
  - Multiprocessor Architecture.
  - Memory architectures.
  - Reconfigurable Architectures.
  - HW/SW Architecture.
  - Low power architectures.
  - Etc.
- APPLICATIONS
  - Case studies from users in industry, academic and students will be high prioritised!
  - HW/SW Component presentation.
  - Prototyping.
  - Etc.
- SURVEYS, TRENDS AND EDUCATION
  - History and surveys of reconfigurable logic.
  - Tutorials.
  - Student work and projects.
  - Etc.

Please contact www.fpgaworld.com/conference