THE UNIVERSITY OF ADELAIDE AUSTRALIA

Exploring the Communcation Architecture in Multi-Processor System on Chips

Final Report

Joel Stanley
1105806

Supervisor: Dr Braden Phillips
Co-supervisor: Assoc. Prof. Michael Liebelt
Advisor: Dr Andreas Hansson

Abstract

Consumer demand for electronic devices that provide complex functionality, operating on large amounts of data, whilst able to run on battery power for days between recharges is increasing. It is common to produce a device that performs radio frequency baseband coding, music playback, video decoding, and gaming all on the one die. These devices employ multiple processing elements running at modest frequencies working together to provide the required processing power with a modest energy consumption. The communication between these processing applications, many of which share a common signal processing structure, is as important as the processing power for the resulting performance of the system.

Application programmers must fully comprehend the system down to its physical properties in order to exploit the hardware to it's full potential. In order to simplify their task, this project intends to both abstract the communication system can be behind a set of useful Application Programming Interfaces (APIs), and characterise it through documentation and benchmarks. The programming interfaces implemented provide inter-process communication for tasks that run on different processes, whilst also providing efficiency and correctness. Efficiency is achieved through placing data close to consumers, and correctness is ensured through data consistency with respect to program order.

This project examines communication paradigms and interconnect architectures available to a multi-microprocessor system running on the Xilinx ML605 Vertex-6 FPGA platform, using the Microblaze soft processor core as the major processing element. The access latencies are quantified in order to illustrate the importance of the communication subsystem on application performance. Finally, the project implements a audio-visual demonstration application that exploits a through understanding of the communication system in order to meet real-time deadlines.

1 Introduction

This project uses the Xilinx ML605 development system to implement a Multi-Processor System on Chip (MPSoC) consisting of multiple processing elements and non-uniform memory structures connected via a arbitrated bus. The MPSoC architecture is common place in the market, with popular commercial designs such as Texas Instruments' OMAP[9] and Sony/IBM/Toshiba's Cell BE.

The ML605 contains a Virtex-6 Field Programmable Gate Array (FPGA) capable of accommodating tens of processors and other Intellectual Property (IP) blocks, along with external memories, connectors and peripherals. Most hardware will be realised in the form of synthesised Hardware Description Languages (HDL) provided by Xilinx, however there will be some custom designs implemented.

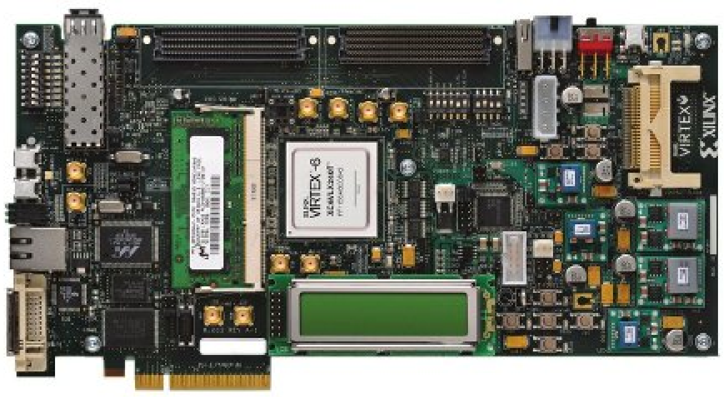

Figure 1: Xilinx ML605 Development Board

The first stage of this project called for a study[25] of the interconnect and memory types available to a designer using the ML605, followed by a study of the communication schemes that are suited for a chosen set of interconnect, memory and application. The study seeks to verify the memory hierarchy that is commonly accepted in computer architecture with quantifiable data, which answers the question of how large is the penalty incurred when using remote memories. This work is unique in the context of the ML605 system, which is new to the marketplace and does not have any independent published benchmarks.

This document details the components available, and then mentions how they are configured for the systems that were built. This is followed by a brief overview of the hardware and software developed for the demonstration systems, and finishes off presenting the results of the memory subsystems that were studied.

A Xilinx ML605 development board was configured with multiple systems consisting Microblaze (μblaze) soft processor cores and three distinct categories of memories; local, remote and off-chip. These memories are constructed using the FPGA's Buffered Random Access Memories (BRAM) and the off-chip Double Data Rate-3 Synchronous Dynamic Random Access Memory (DDR3 SDRAM). The memories are connected over 3 different interconnect paths; the Peripheral Local Bus (PLB), a shared memory mapped interconnect; the Local Memory Bus, an exclusive bus for connecting a memory controller to the μblaze, and the Fast Simplex Link, a unidirectional point to point streaming interface.

The combination of memory hardware and interconnect type dictate the size, access latency, throughput and programming interface of each memory, and it is these properties that form the problem space that is examined. The systems constructed are designed to provide a platform for investigating the problem space, and are used to obtain quantitative data on latency, as well as exploring programming models appropriate for distributed memories.

Using the knowledge gained in characterising the memory and interconnect behaviour of System on Chips implemented in the ML605, an application was implemented that provides efficient decoding of still images using a multi-processor implementation of the JPEG image decoding algorithm. The impact of high latency memory accesses was compared to lower latency access, verifying the conclusions from the first stage of the project outside of basic synthetic benchmarks.

The final stage of the project was to implement a complex demonstration application. The application was selected to include real-time elements that could be easily observed when deadlines are not being met, involving a modest amount of computation whilst keeping the emphasis on communication. A GameBoy emulator was the final choice; keeping the frame buffer updating regularly, and producing sound samples at the required rate are the observable real-time elements. In order to meet these requirements three processing elements was required, plus a number of HDL IP blocks. The communication consisted of a mix of streaming and memory mapped access, and through selecting the right interconnect for the software, allowed the concepts illustrated in the first part of the project to be realised.

2 Related Work

Evolution of FPGAs in recent years has brought a diverse number of prototyping boards that can be used to implement multi-processor systems on chip. A popular board is the Xilinx University Program XUP-V2P, a Virtex-2 based system with a hardware PowerPC core as well as the ability to implement Microblazes. Notably this board was used to prototype the MIPS project[22] collaboration between the University of Adelaide and Harvey Mudd College in 2007, which used two XUP-V2Ps as simulation of a CPU and memory subsystem. As the ML605 that our project uses is relatively new, there are no published papers that relate to at this time, there are some designs done using the predecessor, the ML506[15]. Thus the exploration of the board's capabilities presents a unique contribution to the field of MPSoCs.

There are a number of options when considering what processing elements to use in a MPSoC design. The Virtex-2 FPGA gives designers the choice between Microblaze soft cores and PowerPC hard cores. There is no hard core on the ML605, so no further research on using hard cores was entered into. Other soft cores that have been evaluated, notably the free to use OpenRISC and LEON SPARC-based processors[16]. Whilst these options provide the opportunity for more powerful processing elements, this project is focused on the communication aspects and thus only Microblaze was chosen to be used.

In [10] the author looked looked at the communication architecture of a Microblaze based system in a similar approach to our project. At the time of publishing in 2005, the preferred bus was the On-Chip Peripheral Bus (OPB). This takes the same role as the PLB does in our project, however, they did not benchmark this bus, choosing instead to focus on the FSL links.

The use of FSL was a popular choice in many designs, with [26] using FSL as a means to connect many processors together via a arbitrating IP block. This serves to alleviate the lack of flexibility in connecting processors directly with FSL links. In [23] FSL was used to offload hot-spots in the algorithms implemented on a Microblaze system to a VHDL blocks. They note that applications are best served by offloading small, simple operations to HDL blocks, with larger HDL blocks showing reduced benefit for the logic elements they occupy. The use of FSL as a communication FIFO was demonstrated in [27], where FSL provides distributed memory as well as communication between processing elements.

Communication APIs for embedded systems were surveyed, with the dominant abstraction being message passing[13][20]. Of particular note was COMPSOC[8]; a light weight operating system that uses a message passing API designed for systems on chip with distributed memories named C-HEAP[18]. In [14], an extension of message passing called shared messaging is presented, where the requirement for performing the copy into and out of the FIFO is removed by instead passing pointers to a buffer.

3 Components

This section outlines the peripherals and memories available in the Xilinx XPS library for the ML605. The information is generally applicable to any system designer, however it is presented as background for understanding the systems described in the following section.

3.1 Xilinx Components

The components in this section are available for use with the Xilinx XPS 12.2 development environment. Most are available with earlier and later versions, but details may change between revisions.

3.1.1 Microblaze

The Microblaze[12] is a Reduced Instruction Set Computer (RISC) processor that uses a 32-bit big-endian instruction set. It is a soft processor core, meaning that it is described by HDL and realised by programming it into a Xilinx FPGA. There are a large number of build time configuration options, so only those used are mentioned. It has a 5 stage pipeline and optional shift, multiply and divide instructions, with most instructions completing in a single cycle once the pipeline is full. It has 32 registers, and separate instruction and data memory spaces that can be attached to memories local to the processor via a low-latency Local Memory Bus (LMB), or shared across a shared Processor Local Bus (PLB). The memories themselves may be the same physical storage for both instructions and data, or discrete units.

The μblaze is a PLB master, as well as containing up to 16 Fast Simplex Link (FSL) interfaces, for communication with peripherals, memories and other μblaze. The processor's 32 bit address space allows 2^{32}, or 4GB, of peripherals and memories to be mapped, with the instruction and data memories to be overlapping or separate.

Experimentation with 150, 175 and 200MHz was carried out, with the upper two frequencies unable to meet timing for systems that contain MPMC, the DDR3 SDRAM controller. In this section we assume the device is clocked at 100MHz. This is the default for systems built on the ML605, and is reasonably conservative allowing timing to be met, whilst still providing reasonable performance.

3.1.2 Buffered RAM

Buffered RAM, referred to as BRAM, is the on-chip memory provided by the FPGA. It is used in the system for instruction and data memories, a line buffer in the display controller and as a FIFO buffer for the FSL. The 416 BRAMs available on the given FPGA come in 36 Kbit blocks, for a total of 1872 KBytes. Thus it is a relatively scarce resource and can only be used in increments of 36Kbit, regardless of the actual required storage. Each block is dual ported, meaning it can be used by two different memory controllers at once.

The BRAM presents one cycle of latency for reading one word from a memory location when connected to a μblaze via a Local Memory Bus and controller. It may also be connected to a XPS PLB Controller, a PLB slave to provide an addressable shared memory. The controller has advanced features for transferring data in fixed length bursts of multiple words, however only the single word data beat transfer is enabled in the designs tested, due to the μblaze being unable to use this feature.

3.1.3 Double Data Rate Synchronous Dynamic RAM

There is one DDR3 Small Outline Dual In-line Memory Module (SO-DIMM) installed in a socket on the ML605. With I/O pins being expensive in SoCs, it is common to see only one external memory. The DDR provides 512MB of addressable memory, accessed through a 7-port controller known as the Multi-Port Memory Controller (MPMC). The controller's ports can be individually attached to the PLB, with a single PLB connection chosen for the benchmarked system. This was not done, as each port has it's own buffer and does not guarantee ordering of reads and writes between ports.

As with all dynamic memories, the controller must refresh the values stored periodically to ensure their integrity. Whilst the device is actively reading or writing, this refresh can be delayed for a number of cycles before it will happen regardless of pending traffic.

3.1.4 Fast Simplex Link

The FSL is an address-less streaming interface used to connect two devices, with each unidirectional point to point link specified to be a master or a slave. The master can place one word of data per cycle on the interface, and can optionally block until it is read by the slave. The μblaze uses pairs of FSLs to enable bidirectional communication, using of a specialised `put` and `get` instructions to perform read and writes. In the case of the μblaze, the transfer is limited to the contents of a single 32 byte register. The FSL can be configured to buffer up to 16 words to optimise non-blocking use of the links.

3.1.5 Local Memory Bus

The LMB is used to connect BRAMs to the μblaze cores, enabling single cycle latency memory accesses. This is guaranteed as the bus is exclusive and separate for both instructions and data (thus an instruction fetch will not overlap with a load), so there is no contention from other components. It supports a single master and slave at each end, with the master being a μblaze and the slave a LMB BRAM Controller. The memory types it supports are instruction and data memories, as well as caches. Caches are not used in the benchmarked system.

3.1.6 Processor Local Bus

The PLB[11] is the general purpose interconnect for global communication, supporting a maximum of 16 masters 16 and slaves. Traffic flow is controlled via a hardware arbitrator which can be configured to use round robin or priority based scheduling. Bus masters can perform 32, 64 or 128 bit transactions. The bus contains separate read and write data paths, allowing both a read and write transaction to occur at the same time. The configuration chosen for the benchmark system used 32-bit transactions and round robin based scheduling. All devices are slaves, with the exception of the μblaze cores and TFT controller. Latency is highly sensitive to the amount of traffic on the bus, and thus is variable depending on what components of the system are active.

3.1.7 Timer

The timer is a soft peripheral that is accessed via a memory mapped interface over the system PLB. The timer counts system clock cycles, the same clock that is attached to the μblaze cores. It is included in the system to enable timing of internal events, particularly latencies of the memories and their interconnects. The overhead of enabling and disabling the timer consists of instruction delay and PLB bus access delay, which makes timing variable based on traffic levels on the PLB.

3.1.8 UART

The UART IP provides a serial standard output to the μblaze through a PLB attached memory mapped interface. This was necessary for examining the state of applications running on the μblaze as well as extracting collected data from the system. It is configured to operate at 9600 baud. As it is memory mapped, any processor can use it, however there is no way to discern which output comes from where. The IP contains separate 16 character input and output FIFOs for buffering data.

3.1.9 Clocking

Most peripherals including the are clocked at the same speed as the PLB bus, which it self is clocked at the same rate as the μblaze. Devices that interface outside of the FPGA are exceptions. These exceptions are the Thin Film Transistor (TFT) Controller, which is clocked the pixel clock relevant for the display resolution, and the Multi-Port Memory Controller (MPMC) runs at 200MHz for the 400MHz DDR SDRAM, the I2S audio stream which is clocked at 11290323Hz. The NES controller has no specific clocking requirements, and uses the 65MHz TFT clock.

3.2 Custom Components

Due to limitations of the Xilinx IP, the ML605 development board, and requirements of the systems implemented during the project, a number of custom components were built.

3.2.1 USB I/O Device

The project required a method of moving data in and out of the system that was less restrictive than the UART, which relies on the software running on systems to provide I/O. A system that provided I/O with arbitrary memory space, and importantly initiated by the host, was built with a μblaze and the Cypress CY7C67300 EZ-HOST USB IC[5].

This IC, provided by the ML605, is programmed such that the ML605 enumerates as a USB slave when connected to a PC. The μblaze and EZ-HOST are programmed with software that allows memory read and write transactions to be initiated by libusb[2] application running on a PC. Read and write access can be performed at any location in the μblaze's memory map. In addition to this Memory-Mapped Input/Output (MMIO) mode, a First-In First-Out (FIFO) mode is provided where the data is sent across the μblaze's FSL to a second slave μblaze.

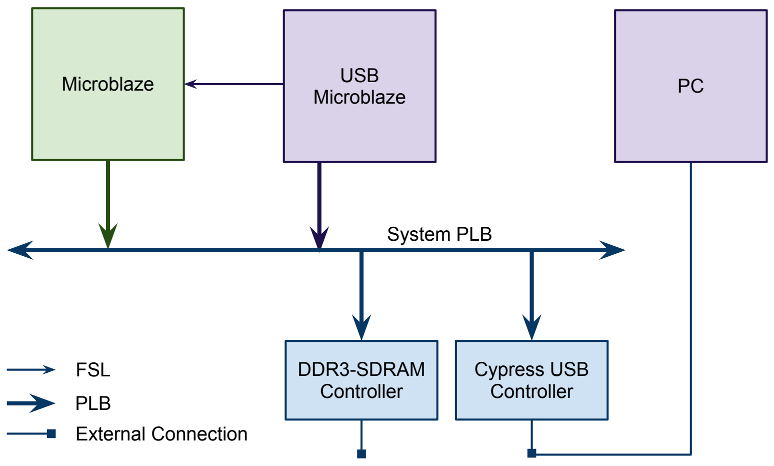

Figure 2: USB to PC I/O System

The combination of the EZ-HOST IC, μblaze, and libusb host software is considered to be the one tile for the purpose of this project. A C code example of connecting to the USB interface and performing a memory mapped read of size 10 from location 0x50000000, a streaming write, and memory mapped write to address 0x51000000 is shown in Listing 1.

```
#include "nestusb.h"
int main() {
  struct libusb_device_handle* udev;
  struct libusb_interface_descriptor interface;
  int buf[10];
  udev = nest_usb_get_device();
  interface = nest_usb_claim_interface(udev);
  mmio_read(udev, &interface, 0x50000000, 10, &buf);
  fifo_write(udev, &interface, 5, &buf);
  mmio_write(udev, &interface, 0x51000000, 10, &buf);
  nest_usb_close(udev, &interface);
  libusb_exit(ctx);
}
```

Listing 1: USB Host Example.

3.2.2 Audio Output

The ML605 lacks a audio peripheral, limiting the boards use as a multimedia demonstration system. It does have a number of GPIO pins available, at voltages ranging from 1.8 or 2.3 LVCMOS, and a 5V level shifted CMOS output. The 5V was chosen as it also includes a 5V and ground connection on the same header. A Cirrus Logic CS4344 24-bit stereo digital to analog converter was chosen, as it was available through our supplier and could be operated with 5V logic.

The CS4344 receives audio samples encoded in an I2S stream. Similar to I2C, the producer supplies a clock on one pin and the data on another. There is also a left/right select pin, to tell the DAC which channel to output the samples, and a system clock. The CS4344 requires a master clock frequency that is a multiple of the desired left/right clock (Table 1 from [4]). For the FPGA to be able to generate the required clock frequency, an approximation had to be made otherwise the combination of clocks required for all peripherals was not achievable. The chosen target frequency is 11.2896MHz, and the achieved frequency is 11290323Hz. This is 6.4ppm off the target, an acceptable margin of error for this application.

A Verilog module transforms 32-bit audio samples into a I2S stream, with the appropriate clocking information. The 32-bit samples consist of 16 bits of right channel data in the most significant bits, and 16 bits of left channel in the least significant bits. The 16-bit numbers themselves are little endian signed samples, with LSB at position 0. The Verilog was re-used from an existing I2S project, but wrapped in VHDL in order to expose it as a FSL slave connection in order to interface it directly to the Microblaze.

There were a number of applications that used the audio output system.

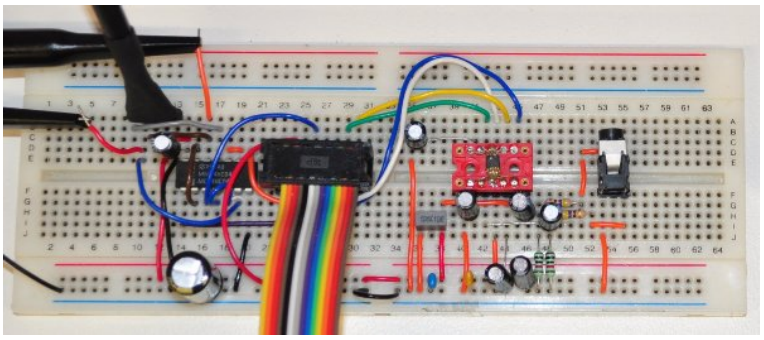

Figure 3: I/O Breakout Prototype: NES controller (Left), ribbon cable to FPGA, Audio output (Right)

- Initially, it was tested with ADPCM samples streamed from a PC.
- A software RTTTL decoder, commonly known as a ringtone decoder seen on early 00's mobile phones, was also ported.
- The initial stages of a MP3 decoder port was begun, but not completed. This application would be suitable for study in the future.
- The 4-bit sound generated by the GNGB GameBoy emulator was output.
- The audio output was combined with some visualisation, in the form of a fast Fourier transform implemented in C on a μblaze. This was outputted to the frame buffer and provided a pleasing picture to accompany the sound output. Due to the limitations of a single μblaze, only a 128-point FFT was achievable in real time.

The audio HDL and schematic is suitable for use by others who wish to have audio output from the ML605. It is intended that the design will be made available to those who wish to use it, with researchers from the University of Twente in The Netherlands expressing interest.

3.2.3 Input Peripheral

Applications running on the ML605 were limited to the 5 pushbuttons mounted on the surface of the ML605. It is not desirable to over-use these buttons, nor to let the public touch the board directly at demos. Once the GameBoy was ported, there was also the need for a input mechanism that had more than 5 buttons. It was decided to obtain an original Nintendo Entertainment System (NES) controller, and interface this with the board, as it had the correct button layout for controlling a GameBoy and is straightforward to interface.

The NES controller speaks a custom serial protocol over a 5V TTL signal. There are two inputs that must be driven, plus ground and power, and the NES controller will output the buttons serially in a fixed sequence on another pin. The signals are depicted in Figure 4.

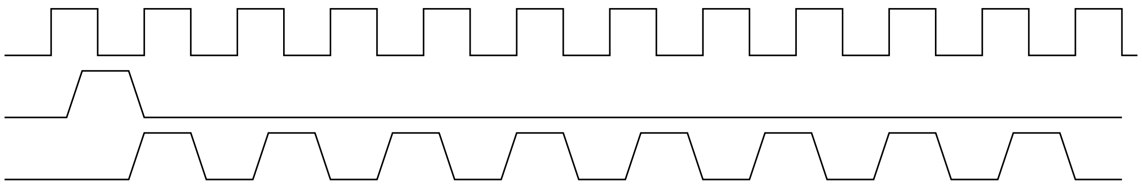

Figure 4: NES Controller Signals: Pulse, Latch, Data

The required signals to query the buttons are generated by a VHDL state machine, with the output packed into a bitfield and placed on a FSL slave port. This allows the button state to be pushed to a consumer, reducing the cost of reading the state from a bus delay to a non-blocking poll of the FSL register and testing if the read was valid.

In testing the state machine, Xilinx iSim was used to verify the correct operation. iSim is a HDL simulator capable of performing behavioural, functional and structural simulation and is free for use with Xilinx ISE. The simulation allows examination of the state machine's internal signals, as well as the external signals. Figure 5 shows the device coming out of reset, and producing the internal clock signals which drive the transitions of the `nes_pulse` and `nes_latch` signals. Latch is held high for two clock

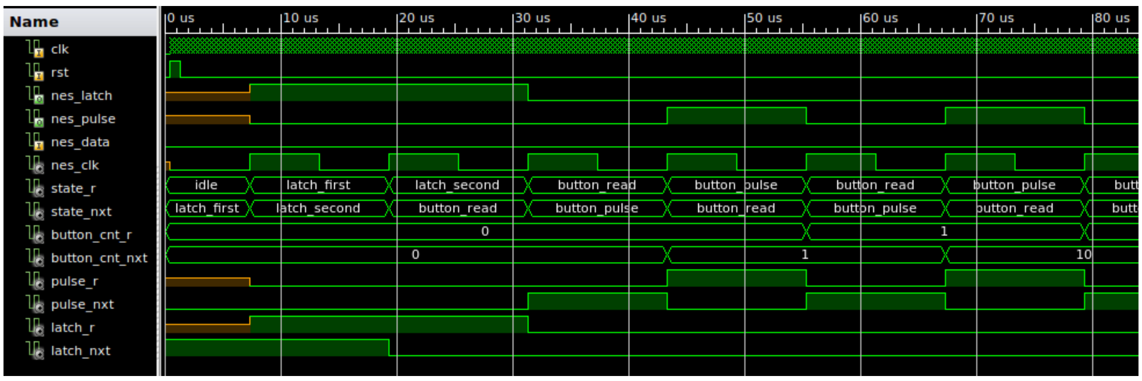

Figure 5: NES Controller State Machine Simulation in iSim

periods, before pulse is transitioned every period. On the rising edge of the pulse signal the state of each button is read from the `nes_data` input. The sequence of button states is A, B, Select, Start, Up, Down, Left Right, with a low signal on the data line indicating the button was depressed. It is not depicted in the figure, but after the state of all 8 buttons has been clocked in, the values are combined into a bitfield and placed on the FSL slave.

The NES controller's connector is not easy to procure, so an extension cable was cut in order to enable easy attachment to the ML605 via the same breadboard as the audio system. The ML605 produces 5V CMOS logic via a level shifter, and the NES controller was designed to produce 5V TTL logic. Due to the difference in logic types, the `nes_data` signal was not received reliably. By placing a Fairchild Semiconductor MMC47HC04 TTL inverter between the ML605 and the controller, the logic levels were driven strongly and `nes_data` could be received reliably, as can be seen in the oscilloscope capture in Figure 6.

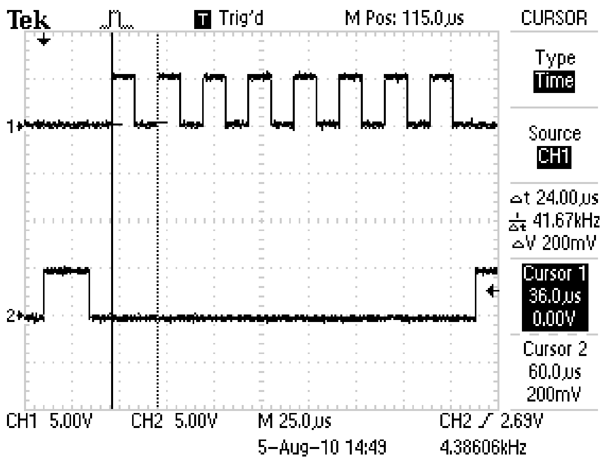

Figure 6: NES Controller Signals on an Oscilloscope: Latch (output), Data (input)

When in use by the GameBoy Emulator system, the IP's FSL was attached to the main processor, which reads the button state once every frame. To appreciate the benefit of having the controller state delivered into the processors register file, the latency of the PLB should be examined. Doing a PLB read takes about 15 cycles, where as the FSL places a word of data directly in the register space of the μblaze. Therefore, the FSL interface is not only simple in terms of HDL to write, it is also efficient.

3.2.4 XGA TFT Controller

The TFT controller is responsible for reading data from a frame-buffer located in our case in the PLB-attached DDR memory. This data is clocked out as pixels to the Chrontel display IC, including the appropriate VGA control signals such as front and back porches. It creates a large amount of traffic on the PLB bus, as to ensure a 1024 × 768@60Hz display is updated without tearing, it must transfer data at 190 MB per second.

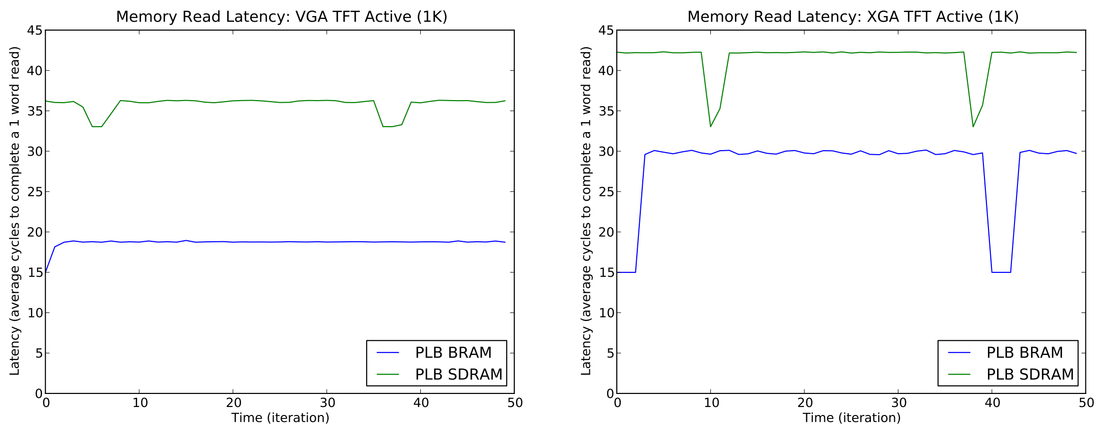

Figure 7: PLB read latency in the presence of XGA and VGA TFT generated PLB traffic

$$\begin{aligned} R &= \texttt{lines} \times \texttt{width} \times \texttt{bytes per pixel} \times \texttt{refresh rate} \\ &= 1024 \times 768 \times 4 \times 60 \\ &= 135\texttt{Mbyte/second} \end{aligned}$$

The provided IP for interfacing with the ML605's Chrontel CH7301C DVI[3] Transmitter is only capable of displaying a fixed VGA resolution of 640×480@60Hz. As there are a number of applications that have been developed for the XGA resolution, 1024×768@60Hz, a number of hours was spent upgrading the hybrid Verilog and VHDL model to output the higher resolution.

Initial work consisted of modifying the relevant counters and clock frequency in order to produce signals with the correct front porch, back porch, and sync pulse timings. The IP uses an internal BRAM as a line buffer, and it fetched a lines worth of data using a PLB master from the SDRAM. The address calculation had to be modified for the larger frame, which was complex as it required adjusting the bit width of a number of signals in order to address the larger width and height of the frame buffer. Finally, the internal BRAM itself needed to be enlarged in order to accommodate the larger line.

The larger frame size also increases the bandwidth that the TFT controller uses, both as it fetches lines to clock out, and also as applications must produce more pixels to fill the frame buffer. This results in increased PLB traffic which affects the performance of other devices on the PLB, as can be seen in the memory latency graph in Figure 7. The bandwidth requirements are 190Mb/s for the XGA frame buffer, and 74MB/s for VGA. Thus for a 2.5 times increase the bandwidth being consumed, we observe a 60% increase in the read latency for the PLB BRAM when moving from VGA to XGA.

4 Inter-process Communication

Software abstractions can be used to lessen the impact of memory latency, and hide the complexities of memory layouts that are non-uniform. There are two broad categories that these abstractions fall under; message passing and memory mapped access.

In systems where there is more than one hardware or software entity must communicate care must be taken to ensure the communication occurs in the correct order, as well as in a consistent fashion. Order is important as most data is dependant on sample n before sample $n+1$, such as the bytes in a serial input stream. Memory consistency is necessary as data cannot be modified by anyone else whilst a consumer is reading it.

Memory mapped interfaces are a traditional view of memory, where an array of addresses allows access to peripherals. The memory is accessed implicitly via pointer manipulations variable accesses. As this is the normal way to access memory, software remains highly portable across systems - providing a program is accessing the correct location in memory, it does not need to be re-written to run on a new system.

Message passing is where tokens (messages) are shared by explicitly using `send` and `receive` interfaces. Message passing provides separation of communication and computation concerns, as well as allowing static analysis and verification of a set of tasks due to the well defined interaction points[14]. These well defined interaction points can be implemented by different systems depending on the nature of the application. Examples are a streaming hardware interface, or a shared memory mapped into the consumer and producer memory space. The downside of message passing is that it is not native to most programming languages, hence programs must be ported.

Communication can be roughly placed into two categories; streaming transfers, where tokens are passed between processing entities, in order, and not read back by the consumer; and memory mapped communication, where both producer and consumers access tokens in any order, and inter-token access is permitted. In terms of programming models, streaming transfers map well to First-In-First-Out (FIFO) patterns, whilst memory mapped communication is a description of a Memory Mapped Input-Output (MMIO) scheme.

Consistency and in-order communication can be provided by hardware, or by software, or by a combination of both. The correct choice of the underlying hardware, coupled with a correct and efficient software API, is a goal of my project. It will assist the team producing the JPEG Picture Frame Muti-Processor system. For instance, in the case of JPEG decoding, the transfer of Minimum Coded Unit (MCU) from the inverse Discrete Cosine Transform (iDCT) to the colour space converter is a streaming transfer - the tokens, MCUs, are produced by the iDCT and are not accessed again. They must retain ordering so that the decoded image is correct. There are similar patterns used in many signal processing applications, comprised of tasks that pass messages between them.

The message passing abstraction is a common one in embedded systems, as it allows a common API to be implemented on top of remote or local memory mapped memories, atop a streaming interface, or even between threads that share an address space in the case of a pthread based implementation. C-HEAP[] provides a solution to the bounded buffer consumer/producer problem, where a consumer is placing data into a buffer of finite size and a consumer is removing data.

A working implementation of a message passing interface is C-HEAP[18]. It ensures release consistency by using a administration data structures to arbitrate the access to a memory mapped FIFO. Alternatively, it could be mapped to a FSL link in a μblaze. It has been used by Hansson in CoMPSoC[8] and other projects, and I have adapted his C implementation to run on both a single-process `pthreads` simulation on a PC, as well as with two μblaze units using shared memory. So far only trivial demonstration applications have been used, but it is planned to port a two core JPEG example to the system.

A variant on message passing is *shared messaging*, presented by Kiran in [14]. The modification is to use message passing to exchange synchronisation tokens, whilst the actual data is simply reserved regions of memory. These tokens provide read or write permission to the holder. This eliminates the need to perform the copying of data from one location to another as in C-HEAP. The paper only describes results from a functional simulation written in C++, which indicates that measuring performance of a hardware implementation would provide an avenue for original experimentation.

5 System on Chip Architectures

When arraigning the processing elements in a multi-processor systems on chip, there are two approaches commonly used. The first is arrange heterogeneous processing elements, with each having a dedicated specific functionality, into an arrangement that suits the communication requirements of the combined applications.

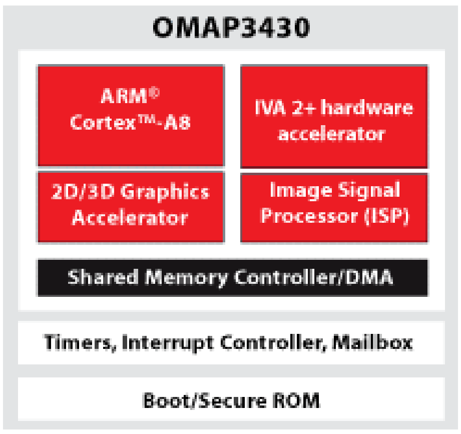
OMAP3430
ARM® Cortex™-A8
IVA 2+ hardware accelerator
2D/3D Graphics Accelerator
Image Signal Processor (ISP)
Shared Memory Controller/DMA
Timers, Interrupt Controller, Mailbox
Boot/Secure ROM

Figure 8: OMAP3430 SoC Block Diagram

An example of this is the OMAP3430 from Texas Instruments. It is a mobile SoC that includes four main processing cores; an ARM applications processor, the IVA video and audio codec accelerator block, a POWERVR SGX graphics accelerator, and a image processing core. Each core has a predetermined functionality for a specific set of tasks. Whilst those tasks are not being executed by the system, these cores must lay idle, as a traditional graphics accelerator cannot be easily re-tasked to perform video acceleration. Data flow between each core is also determined statically at design time, meaning all interconnects must be dimensioned for worst-case throughput without allowing the application to manage resources at run time. This lack of flexibility means interconnects may lay idle whilst other parts of the system are at peak capacity.

The other approach is to place a number of homogeneous cores connected to an interconnect. The cores can be re-purposed for whatever tasks the application requires at that time. It allows software to dictate the mappings between each processing element, with the disadvantage that the shared interconnect may become overloaded as all elements are contending for the same bandwidth. The complexity with this approach is that the computation must be structured such that the workload can be distributed across many processors. In this situation applications can be limited by the communication between processors; efficient computation requires loosely coupled workloads and a easy to use but low latency communication architecture. This has proven to be a challenge[17], as demonstrated by the Intel Larrabee[24] architecture, and the Cell BE[21] as used in Sony's Playstation 3.

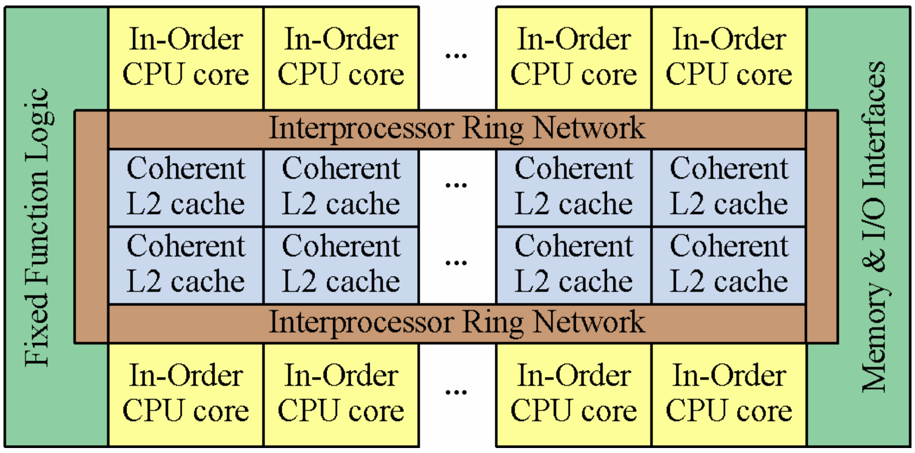

Figure 9: Larrabee Block Digram

This second approach is referred to as a tiled architecture. It has been used in Intel's Larrabee, IBM/Toshiba/Sony's Cell BE and NXP Semiconductor's Nexperta, and will be utilised by this project. The

Larrabee consists of a number of CPU cores connected by a bi-directional ring network. The ring network also provides access to memory, IO, and a large level 2 cache. Each processor is allocated a sections of the cache. A processor's cache section is faster, in terms of latency, than the rest of the cache. In this sense it acts like a local scratchpad memory, as well as a cache for off-chip memories. The Cell BE contains 8 tiles which themselves consist of a vector processing unit and a 356KB of local memory. Tiles are connected to a main processing element via a shared bus, which the tiles can use to fetch data by programming a DMA. The tiles cannot access memory directly, which enables the DMA to manage bandwidth of the shared interconnect. Unlike the Larrabee where all processing elements are homogeneous, there Cell also contains a powerful main processing element that is a different architecture to the tiles, it can be used as a controller for the tiles, but also perform calculations of it's own.

In this project a tiled architecture is adopted, with each tile containing a μblaze with dedicated instruction and data memories. Memory-mapped interfacing to other tiles and shared peripherals is through a global PLB. FSL provide one-way streaming connectivity to other tiles, peripherals and specialised IP cores.

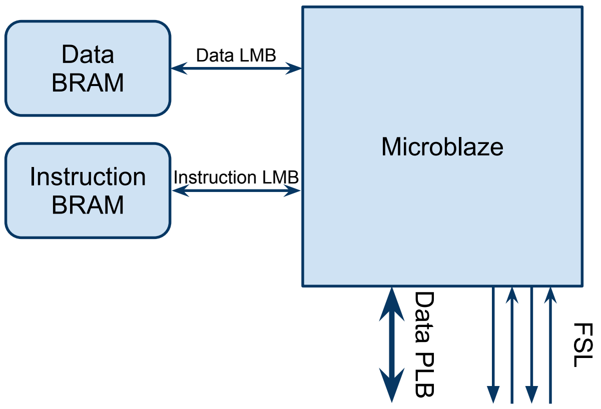

Figure 10: Microblaze Tile

Each tile is assigned a task, and communication links between tiles are provided where necessary. Where possible communication is arranged to stream from one tile to another via FSLs with large FIFOs. Streaming places data close to the consumer, and has no bus overhead - writes are acknowledged by the FSL the cycle after data is placed on the lines, unlike the PLB, which requires writes to be acknowledged by the slave memory controller before the processor may continue.

```
5bc:   e86027f0    lwi     r3, r0, 10224
5c0:   b000dead    imm     -8531
5c4:   30c0beef    addik   r6, r0, -16657  // Load 0xDEADBEEF into r6
5c8:   a0840080    ori     r4, r4, 128
5cc:   d8832800    sw      r4, r3, r5      // Start timer
5d0:   b800000c    bri     12              // Jump to 5dc
5d4:   f8c70000    swi     r6, r7, 0       // Store value in memory
5d8:   30e70004    addik   r7, r7, 4       // Increment pointer
5dc:   16573803    cmpu    r18, r23, r7    // Loop condition
5e0:   bc52fff4    blti    r18, -12        // Conditional branch to 5d4
5e8:   e0e026b0    lbui    r7, r0, 9904
5ec:   30a027ec    addik   r5, r0, 10220
5f0:   10c00000    addk    r6, r0, r0
5f4:   c8643800    lw  r3, r4, r7
5f8:   a463ff7f    andi    r3, r3, -129
5fc:   d8643800    sw  r3, r4, r7          // Stop timer
```

Listing 2: Disassembled Microblaze code for memory benchmarks

6 Benchmarking

A single tile system system with the Xilinx Timer peripheral, MPMC SDRAM controller and TFT framebuffer was developed to characterise and allow experimentation with the ML605's memories. The benchmarks were devised to quantify the estimates made about the memory hierarchy (Table 1). It is known that the SDRAM is quite high throughput but long latency, as it is off-die and must be refreshed as with all DRAMs. The documentation for the PLB suggests arbitration induces a 3-cycle latency[11], and the LMB is supposed to have a single cycle access latency. Along with these assumptions, the ability to perform accurate measurements using the tools available was investigated.

The benchmarks for each memory type consist of C code, compiled with `gcc-4.1.2` with optimisations for small code size, `-Os`, enabled. The code performs a pointer write of a 32-bit word to memory, increment the pointer location, and repeat until the pointer reached a pre-determined maximum value. The maximum value was varied in order to see effects of large versus small accesses. Larger values, ranging from The disassembled code for this loop is displayed in Listing 2. Completion of this entire loop is termed an *iteration* in the graphs, and it is a variable number of words for different graphs.

To characterise transient noise such as PLB traffic, 50 iterations were performed for each benchmark. This period of measurement proved long enough to observe transient effects, without producing an overload of data. An important factor to take into account when viewing the results: the latencies are for a fully active bus performing accesses in the order of kilobytes. For applications that are accessing individual words, it was observed that access latencies will be much longer, as seen in Figure 7.

The writing of data to a memory mapped device indicates that the data has been successfully placed on the bus and acknowledged by the slave device. This means that the write latency is a measure of the cycles that it takes to post the write; the data may not have yet been stored in memory. This is the opposite of the case for reads, which are blocking; when the read instruction has completed the data is available in the μblaze register.

The measured time is illustrated in Figure 11. The number of cycles that occur between the two arrows are what the benchmark is recording. The overhead consists of PLB traffic whilst starting the timer, instruction overhead as the program increments counters and jumps to loop, and the actual delay due to the memory device. Instruction overhead is 4, plus a pipeline bubble due to the branch. There are 6 instructions plus PLB delay when stopping the timer. Each instruction corresponds to a cycle, as mentioned in Section 3.1.1 on the μblaze.

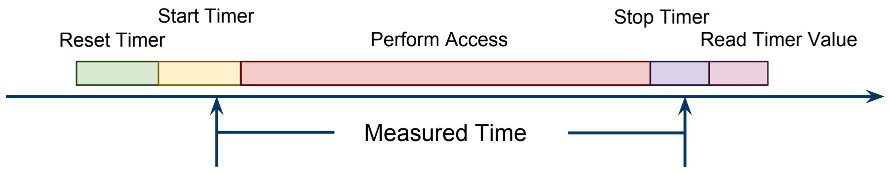

Figure 11: Timeline of a memory benchmark

It was observed that the sending of benchmark results down the 9600 baud UART would create enough PLB activity that it skewed measurements. This could be attributed to the filling of the UART's 16 character buffer faster than it could flush down the serial line. The solution was to write results to local memory and once the benchmark was complete, send them across the UART. This may still create unwanted activity for the local memory tests; a potential solution would be to send the results across the FSL link to another μblaze's local memory.

Table 1: Predicted Memory Performance

Memory	Interconnect	Predicted Latency	Predicted Jitter
Local BRAM	LMB	1 cycle	None
Shared BRAM	PLB	20 cycles	Large
SDRAM	PLB	100 cycles	Massive
μblaze	FSL	1 cycle	Low[1]

6.1 Timer delay

The timer peripheral counts cycles of the global clock, as outlined in Section 3.1.7. The timer is attached to the μblaze via the PLB, which means there is a delay in accessing it. Also, it's operation is effected by, as well as effects, the performance of measuring PLB attached devices. The timer contains a control register for configuration, including the start and stop of the timer. As the timer is attached to the shared PLB, to ensure safe operation the value must be read out, the start/stop bit masked out, and then written back. This means two PLB transactions, as well as an instruction. However it was found that there was considerably more delay - up to 70 cycles.

The following sections discuss the reason for the large overheads in starting and stopping the timer. However, as these delays were measured to be relatively constant - either 25 or 31 cycles, depending on if the TFT was enabled or not - they can be factored out of the measurements made. The details are included as an illustration of the complexity of the system, particularly the inefficiencies in using a high level language like C to simply twiddle bits in a register.

6.1.1 Stop

The assembly code for enabling and disabling the timer is attached in Appendix 3. The `XTmrCtr_Stop` function contains 36 instructions, two of these are the preamble and two the epilogue to setup and tear down the local stack. 32 of the remaining instructions are a series of assertions that ensure the device is present at the location indicated. The actual disabling of the timer is three instructions where the control register is read, the 'on' bit is masked off, and the result written back to the memory mapped register. By removing the unnecessary asserts, and forcing the compiler to inline the code by declaring it `extern static` the disabling of the timer takes 7 instructions, plus the PLB delay. The high number is due to gcc's emitting loads for structures to calculate the location of the timer control structure, which are not used.

```
void XTmrCtr_Stop(XTmrCtr * InstancePtr, u8 TmrCtrNumber)
{
    u32 ControlStatusReg;

    Xil_AssertVoid(InstancePtr != NULL);
    Xil_AssertVoid(TmrCtrNumber < XTC_DEVICE_TIMER_COUNT);
    Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

    /*
     * Read the current register contents
     */
    ControlStatusReg = XTimerCtr_ReadReg(InstancePtr->BaseAddress,
                          TmrCtrNumber, XTC_TCSR_OFFSET);
    /*
     * Disable the timer counter such that it's not running
     */
    ControlStatusReg &= ~(XTC_CSR_ENABLE_TMR_MASK);

    /*
     * Write out the updated value to the actual register.
     */
    XTmrCtr_WriteReg(InstancePtr->BaseAddress, TmrCtrNumber,
              XTC_TCSR_OFFSET, ControlStatusReg);
}
```

Listing 3: Xilinx Timer Driver: Original stop function

6.1.2 Reset

Resetting is not included in the measured period, therefore it is not going to add noise to our measurements. It does take 148 instructions in `XTmrCtr_Reset` plus the setup in the calling function.

6.1.3 Start

Starting the timer was originally 152 instructions. As with the Stop operation, the first 30 were useless assertions, followed by some configuration and finally the starting of the cycle timer. It could be reduced to the three instructions discussed in the Stop operation.

6.2 LMB attached BRAM

Figure 12 shows the local memory read and write time are highly deterministic, and as expected for a device that is not attached to the PLB, unaffected by the PLB traffic generated by the TFT display controller. The latency is expected to be 1 cycle, which does not account for the 6 and 8 for writes and reads respectively. Taking the write case as an example; the extra cycles can be attributed precisely to for loop which performs the write (see Listing 2)

- 4 for instructions
- 1 extra cycle for the pipeline bubble due to the conditional branch
- 1 for the actual latency of the read

And we have the 6 observed cycles. A similar explanation can be provided for the read case, where the μblaze instructions are slightly different.

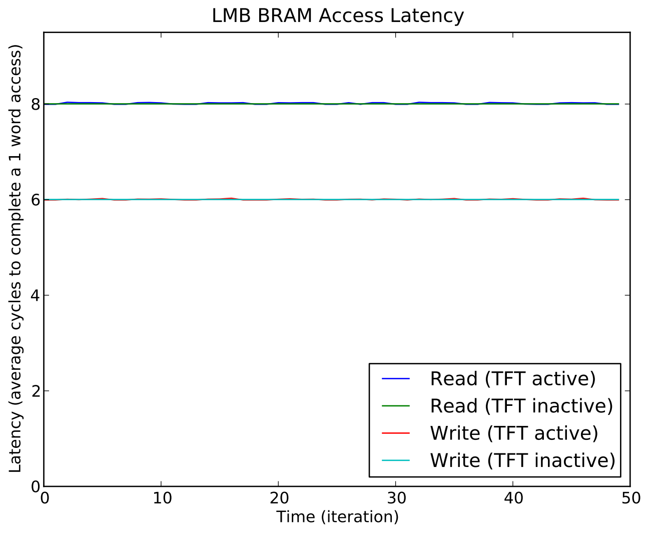

Figure 12: LMB attached BRAM benchmark

6.3 PLB attached BRAM

The PLB BRAM, referred to as shared BRAM as it is not local to any processor, and can be accessed by any device attached to the system bus, was benchmarked with 32 words, or 1KB worth of 32-bit word accesses. The results in Figure 13 clearly show the effects of PLB traffic on the read performance, but are barely discernible for writes. This is attributed to the PLB bus being free of other write traffic, as the TFT controller generates only read transactions. The variation of approximately 5 cycles in read latency is due to the periods where the TFT controller is clocking out the blanking interval, and thus not performing any PLB reads. As the vertical blanking interval is small - approximately 10% of the time spent clocking out pixels - there is not enough time to perform a full iteration of 1KB worth of data before the TFT again produces PLB traffic. It is assumed this is why the latency does not drop down to the average of 15 cycles seen when the bus is completely idle.

6.4 PLB attached SDRAM

The results for the SDRAM benchmarks are presented in Figure 14. The SDRAM is also PLB attached, and is affected by the PLB traffic as expected. The 41-cycle average read latency comprises of both the PLB delay as well as the pre-charge and access times for the DDR3-SDRAM. Write speed is identical for both

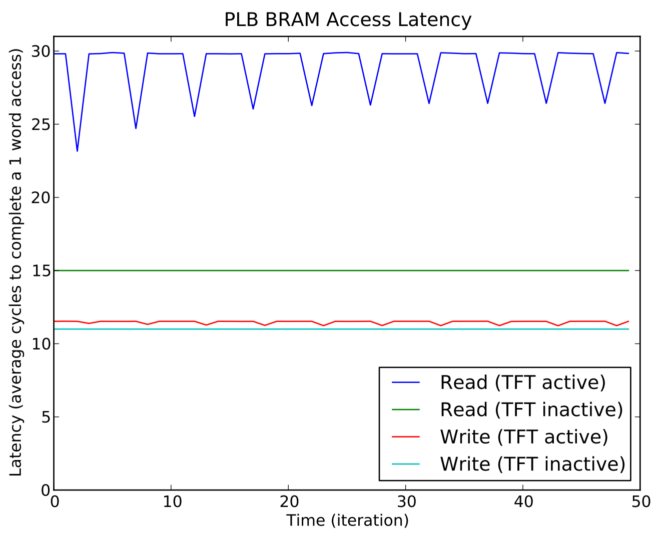

Figure 13: PLB attached BRAM benchmark

the loaded and unloaded PLB due to the split read/write paths of the PLB. It is assumed that the loaded read speed is more constant than the BRAM as the SDRAM write size was 1024KB; much larger than the PLB BRAM's 1KB. Due to this larger access size, the MPMC's ability to pipeline accesses smoothed out the changes in latency that the BRAM's saw due to PLB traffic.

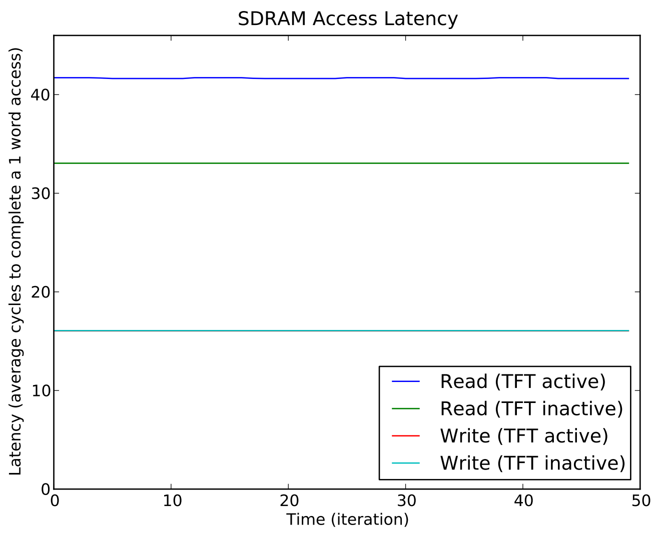

Figure 14: PLB attached SDRAM benchmark

6.5 Comparison

A goal of the benchmarks was to gain a quantified understanding of the latencies in accessing different memory types. The previous section examined each memory type in turn. In this section, the sample size was fixed to 1KB (32 words) and the latency of the three memory types compared under the load produced by an XGA frame buffer.

Figure 15 shows the write performance, indicating that the extra traffic does not make a significant impact on latency. This can be attributed to the lack of write traffic on the PLB. As the PLB has separate buses for reading and writing, one of each can occur simultaneously. The TFT controller generates read traffic as it fetches lines from the framebuffer, but there is no write traffic. Further experimentation with a source of write noise would be required to determine the true factor behind equal latencies.

In Figure 16, it can be seen that the PLB attached devices vary significantly in latency depending on traffic. Table 2 shows the SDRAM is 80% slower and the BRAM is over 50% slower. This is due to the contention produced by the TFT traffic. Also of note is the variability in the PMB latency - there is a 15 cycle difference between the slowest read and the fastest. If a system was being built with a hard real-time requirement on a task with that latency, it would need to take the large value to ensure it meets timing. Guarantees such as this would be better suited to a Network on Chip.

Table 2: Read latency

Memory	Loaded latency	Unloaded latency	Increase
LMB BRAM	8 cycles	8 cycles	0%
PLB BRAM	28 cycles	15 cycles	53%
PLB SDRAM	41 cycles	33 cycles	80%

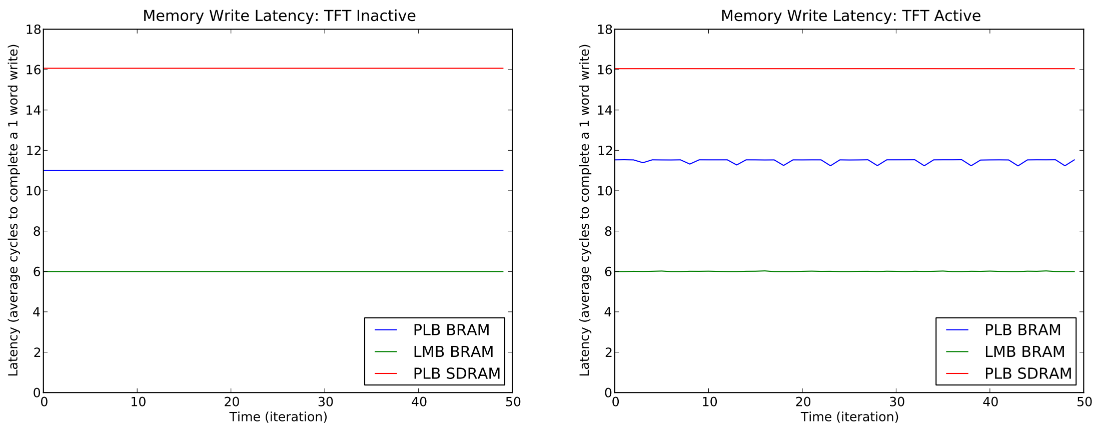

Figure 15: Comparison of write latency

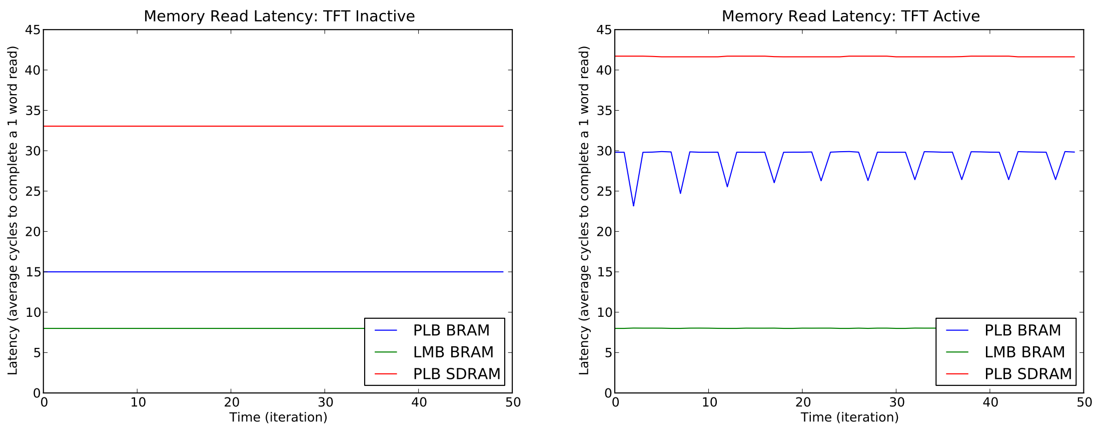

Figure 16: Comparison of read latency

7 Systems

A number of test platforms were developed over the course of the project, beginning with basic functionality and becoming increasingly complex as important components were built and understood. These test platforms enabled the production of two major systems: the three tile JPEG Picture Frame, and the four tile GameBoy Emulator.

7.1 Common blocks

These blocks are built from the components mentioned in Section 3 and are used for building the systems that follow in this subsection.

Local BRAM

Each processor contains it's own local BRAM is used as the instruction and data memories. As the system is a Harvard memory architecture, it uses separate instruction and memory address spaces. These spaces may be mapped to the same physical space in one dual-ported BRAM, however, the benchmarked system uses separate physical devices with an overlapping address space.

Remote Shared BRAM

The remote BRAM is attached to a BRAM Controller which is placed on the PLB bus as a slave. This controller is then mapped into the μblaze address space and can be accessed as a continuous memory device, up to the size of the BRAM. The BRAM can be a dedicated memory, or the PLB controller can be attached to the second port of a μblaze's local memory. By configuring the memory in this way, other tiles can read and write to the μblaze's local memory, and providing it does not disturb data that is being used by code on the local processor, the other tiles see it the same as a remote memory.

Remote Shared SDRAM

The single SDRAM memory module is loaded into a socket on the ML605, and is accessed from the FPGA through the Multi-Port Memory Controller. This IP block contains 6 different ports that can interface over FSL or PLB interconnects. In the benchmarked system, it is connected to the PLB and all 256MB assigned to a single continuous memory space in each μblaze.

8 JPEG Picture Frame

The first application to be developed for this project was a three tile picture frame application, capable of decoding JPEG images taken from a FSL input or SDRAM using two tiles to run the JPEG algorithm.

8.1 Single Tile System

Initially a single processor system derived from the Xilinx demonstration system was used to obtain an understanding of the tool flow. At the centre of the system is a single μblaze, with a single 32KB instruction and data memory mapped to an overlapping address space. The main system bus is the PLB, and it contains a series of low speed peripherals - timer, GPIO, LEDs, push buttons, character LCD, interrupt controller - most of which have not been used by the project. The one port of the SDRAM's MPMC is attached to the PLB, providing 256MB of address space for the processor's general use but more importantly the frame buffer. The frame buffer is held in a fixed location in the SDRAM, and accessed over the PLB by the TFT controller.

The software written for this system was a simple port of the *5kk03*[19] JPEG decoder, with the input image is coded into a static c byte array, and stored in the local BRAM. Also included was as a series of frame buffer test applications. To select between each of the applications a basic menu was provided over the UART. Feedback was provided both over the UART as well as on the TFT display.

8.2 Two Tile System

With the single processor system working reliably, a second tile was added that contained a μblaze, local instruction and data memories, and a FSL connection to the other tile. The second port on the local Data BRAMs was attached to the global PLB, allowing processors to access another tile's data as a remote memory.

This enabled the porting of C-HEAP to the two-tile system, with the shared FIFO living in the consumer's local address space. C-HEAP was able to pass simple arrays of integers between each processor as a test for running a C-HEAP enabled JPEG decoder. The shared BRAM allowed the effects of the PLB to be characterised without the extra complexity of the SDRAM adding noise to the measurements.

Also added to this system is an IP block to enable software controlled reset for the two slave μblaze. This allows slave μblaze to be held in reset whilst initial conditions are set, such as initialising data in the SDRAM which cannot be initialised by the XPS toolchain.

The software was re-written to provide a timed read/write test of the three memories: local BRAM, shared BRAM and shared SDRAM, as well as a benchmark that performed the same writes a specified number of times. This benchmark allowed the observation of transient effects such as SDRAM refresh and PLB activity.

8.3 USB I/O System

The USB system is built from two μblazetiles, one of which is dedicated for communication through the EZ-HOST USB IC as described in Section 3.2.1. The second μblaze tile, termed the slave, is connected to the USB system via FSL links.

Figure 17: Surfer JPEG test image

The slave is loaded with a JPEG decoder allowing a host PC to stream a JPEG image to the system, which decoded it on the fly and displayed it on the TFT. The decoding speed is bound by the JPEG decoder. When decoding the Surfer standard test image (Figure 17), scaled to the size of the 1024 × 768 frame buffer, the decoding speed was found to be 13.6 seconds when using a software multiplier and barrel shifter, and 3.5 seconds when the processor was built with hardware units to do multiply and shift operations. This speed was a measurement of the time for the entire image to be transferred from the host PC over the USB FIFO. This transfer is bound by the rate at which the JPEG decoder μblaze consumes FSL communication, as the FIFO is of a finite size and once it is full the host must wait before sending more data.

There is little performance benefit to be gained from using a second μblaze to do the communication, as the USB μblaze's utilisation is minimal. The system was designed only to prototype host communication over USB, with JPEG decoding as a convenient consumer of the data.

8.4 Three Tile System

The three tile system comprises of the USB I/O tile, and two μblaze tiles with PLB mapped data memories, as in the two tile system. The reset lines of the processing tiles was wired to a `xps_gpio`, a simple IP block that allows memory mapped control of the state of the wires on the block. This allowed the processors to be reset by software, in this case from a PC via the USB tile.

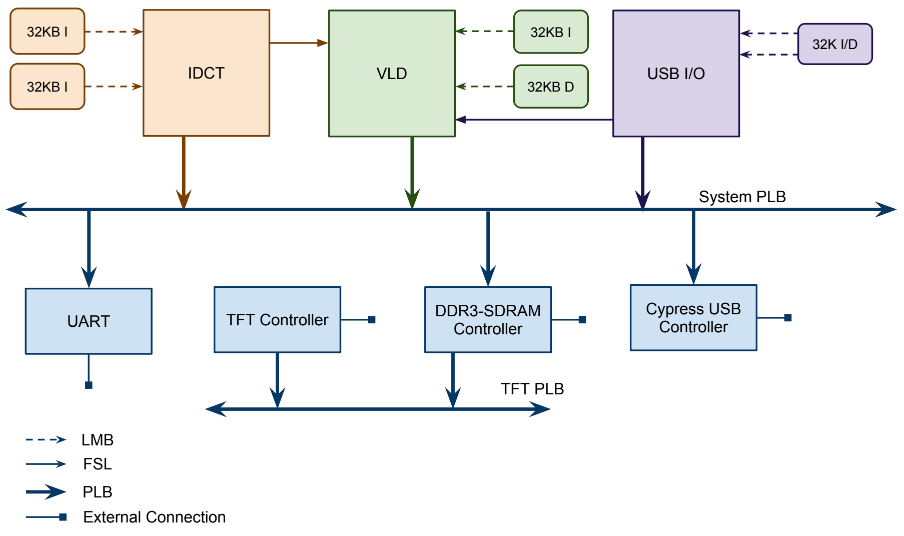

Figure 18: Three Tile JPEG System Block Diagram

The USB tile is dedicated to host communication, in this case it performs one of two functions:

- Streaming JPEG data over a FSL port.
- Writing a JPEG image to the SDRAM.

The functionality depends on how the JPEG decoder wishes to read in the image data.

The two JPEG tiles were configured to use a C-HEAP message passing FIFOs that used memory mapped writes to copy data from the producer to the consumer tile. The C-HEAP administration data was placed in the local memories, with the FIFO placed in the consumer memory.

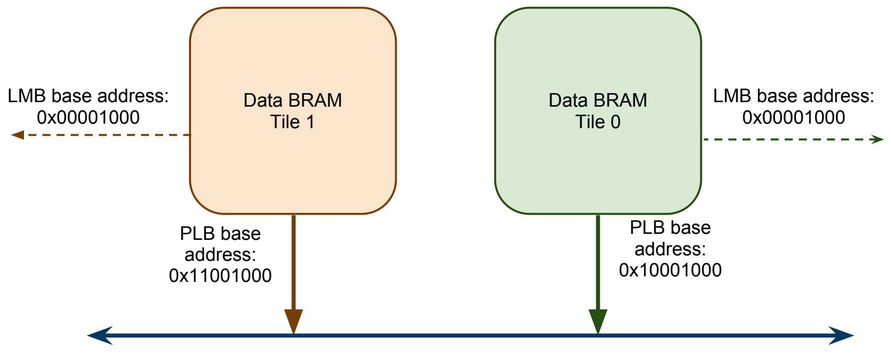
LMB base address: 0x00001000
Data BRAM Tile 1
Data BRAM Tile 0
LMB base address: 0x00001000
PLB base address: 0x11001000
PLB base address: 0x10001000

Figure 19: Shared BRAM Memory Map

The BRAM PLB controllers were configured to map the data memories on the PLB at an offset of 0x11001000 for tile 1, and 0x10001000 for tile 0 as illustrated in Figure 19. The LMB mappings were

0x00001000 for both tiles, meaning that for each core, adding 0x1X000000 to a LMB data memory address would find the aliased PLB data memory address, where X would be the id of the tile.

```
/* Local storage for producer administration data. */
volatile fifo_t L_fifo_adm_prod;
/* Pointer to remote consumer administration data. */
volatile fifo_t* REMOTE(fifo_adm_cons, UBLAZE1_DMEM);
/* Pointer to remote C-HEAP FIFO. */
volatile unsigned int* REMOTE(fifo_data, UBLAZE1_DMEM);

/* Local data that will be written into remote FIFO. */
data_t send_data;

/* Initialise administration data. */
vld_init_fifo(&L_fifo_adm_prod, R_fifo_adm_cons,
    sizeof(data_t), DIM_FIFO_SIZE,
    (unsigned int)R_fifo_data,
    (unsigned int)R_fifo_data - UBLAZE1_DMEM);

/* Await space for performing write to FIFO. */
do {
  write_pointer = vld_claim_space(&L_fifo_adm_prod);
} while (write_pointer == 0);

/* Perform remote write. */
vld_remote_write(write_pointer, &send_data, sizeof(data_t));

/* Release data: data is now ready for consumer to read. */
vld_release_data(&L_fifo_adm_prod, R_fifo_adm_cons);
```

Listing 4: C-HEAP Producer Example

```
/* Pointer to remote producer administration data. */
volatile fifo_t* REMOTE(fifo_adm_prod, UBLAZE0_DMEM);
/* Local storage for consumer administration data. */
volatile fifo_t L_fifo_adm_cons;
/* Local storage for C-HEAP FIFO. */
volatile unsigned int L_fifo_data[sizeof(data_t) / sizeof(int) * FIFO_SIZE];

/* Local data that will be read from the FIFO. */
data_t recieved_data;

/* Wait until input is available from the VLD. */
do {
  recieved_data = (data_t*) idct_claim_data(&L_fifo_adm_cons);
} while (recieved_data == NULL);

/* Notify producer that data has been consumed. */
idct_release_space(&L_fifo_adm_cons, R_fifo_adm_prod);
```

Listing 5: C-HEAP Consumer Example

An example of how the C-HEAP administration data is set up is shown in Listing 4, as well as an example of sending the ficticious `data_t` structure to a consumer, which claims the message and releases the space in the FIFO in Listing 5.

Tile 0 runs the VLD (Variable Length Decoding) portion of the JPEG algorithm which includes inverse zig-zag and de-quantisation, whilst tile 1 runs the IDCT (Inverse Discrete Cosine Transform) plus the colour space conversion and writing to the framebuffer: the naming reflects the majority of the work to be done by each tile. The C code for the decoder has been previously used on a multi-processor C-HEAP system from Silicon Hive[7], and required minor porting to adhere to the memory map of the system built on the ML605.

8.5 Software Toolchain Shortcomings

The Hive toolchain incorporated primitives for placing data in specific memories, including initialising shared memories. As of version 12.2, the Xilinx tools are based on `gcc-4.1` and do not offer either of these features. Therefore, a naming convention was used for data that was to reside in remote meories, allowing it to be extracted after the binary was build using `nm`. As there is a circular dependency - the VLD program depends on variables stored in the IDCT memory, and vice-versa - this extraction process is run once, the programs re-compiled with the new location information, and then compiled again. As the second iteration only adjusts memory pointers by a few positions, the variables stay in the same stored location and hence the programs converge to a steady representation.

Another challenge to solve was how to bootstrap the interdependent processors. As it was not ever guaranteed which order the code would be executed, it was decided a special value known to both programs would be stored in a predetermined location in the SDRAM. This value would cause the IDCT core to enter a busy wait uni the producer, in this case the VLD tile, modifies it once the administration data is in place.

As SDRAM is uninitialised on power up, the USB I/O system was used to place the special busy-wait value in the SDRAM and once this was done, the processors were brought out of reset. The VLD tile will then set up the administration data structures, and modify the mutex value to allow the IDCT core to begin execution.

This complex system could be simplified in hindsight. The VLD tile could be allowed to come out of reset with the rest of the system, and once it's administration data was set up, it could take the IDCT core out of reset.

8.6 Hardware Toolchain Shortcomings

Another shortcoming of the Xilinx XPS toolchain was discovered when attempting to use split instruction/data memories. In order to place the `text` section in the instruction BRAM, and the `bss`, `data`, stack and heap in data BRAM, a linker script is required. This can be generated in XPS.

When generating a system using XPS, the IP modules are synthesised into a netlist. This netlist is then mapped to FPGA elements by a tool called `map`. As of XPS 12.1, `map` is able to run in a multi threaded mode by adding the argument `-mt 2;` to the file `opt/fast_runtime.cmd`. The output of `map` is then passed to `par`, the place and route tool that decides where on the FPGA floorplan the logic elements are used, and how to connect them. This produces a file `implementation/bitstream.bit`, which contains FPGA logic element configuration only. The initialised BRAM blocks are inserted by the `Data2Mem` tool, which programs the correct BRAM blocks with the relevant program data, obtained from the `ELF` binaries produced by the c compiler, `gcc`. `Data2Mem` produces the file `implementation/download.bit`, which is programmed into the FPGA over USB JTAG using the `impact` tool.

A downside of using memories that are mapped into the address space of two programs, such as BRAMs that are mapped remotely over the PLB bus, is that Data2Mem will initialise the BRAMs twice - firstly with the LMB mapped data from the local μblaze's `ELF`, and then with the PLB mapped data from each remote μblaze$\dot{T}$hisis undesirable, as it results in the PLB mapping - which is generally left with no initialised values - winning out, and leaving the data BRAM zeroed out. The work around was to edit `implementation/system_bd.bmm`, which is a mapping of physical BRAM blocks to the address space of each μblaze. `Data2Mem` uses this file to decide what ELF sections go to which BRAM, so by removing references to PLB mapped BRAMs from `system_bd.bmm`, `Data2Mem` does the correct thing - initialise BRAMs with the LMB mapped data, and leave the PLB mapped BRAMs unmodified.

8.7 JPEG Performance

With the two-tile JPEG decoder functional, it was tested with the input JPEG stored in the SDRAM by the USB tile. A series of optimisations was attempted, based on assumptions of the latencies present in the system. The baseline image was decoded in around 300 million cycles. This is the number of cycles from the start of image marker to the end of image, as counted by a timer peripheral. It is not the number of cycles consumed decoding, as there are two processor tiles operating asynchronously - one may be idle, and hence the other will do more work.

The first optimisation was to place the input data closer to the consumer. In this case, that meant using a FSL FIFO to stream the JPEG into the VLD tile. This reduced the decode time by over half by eliminating the 33 cycle (best case) SDRAM PLB fetch latency. As can be seen in Figure 20, further optimisations were not as dramatic. They focused on eliminating computation or memory accesses that could potentially be dealt with by additional hardware. For example, colour space conversion was not performed as it is possible to do this with a IP block. Also, writing to the frame buffer was removed as this could be completed with a DMA unit instead of tying up the IDCT tile.

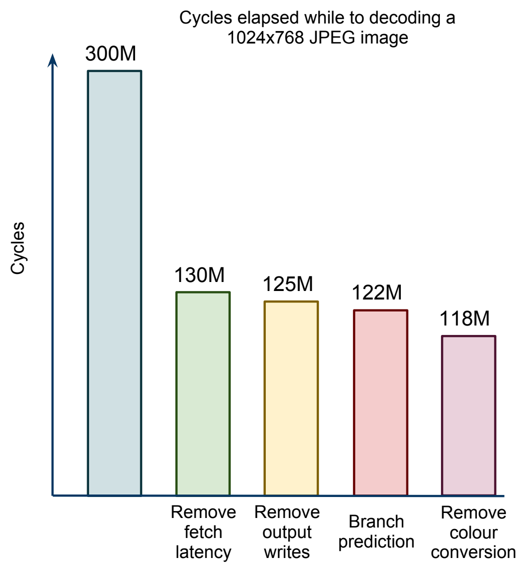

Figure 20: C-HEAP JPEG Decoding Performance

9 GameBoy Emulator System

The second application implemented using a tiled architecture was a GameBoy Emulator. The open source GNGB software was chosen for it's small code size and simple interfacing. It uses SDL (Simple Directmedia Interface), a cross-platform library for abstracting keyboard input, sound and video output.

The GameBoy Colour was an 8.4MHz Z80-like ASIC that produced a 160x144 framebuffer at 5:5:5 colour depth (5 bits per R, G, B component), and 4-bit two-channel sound output[1].

9.1 Software Porting

As this project was focusing on the hardware aspects of the system, a majority of the porting work was completed by Andreas Hansson. The SDL code was removed and replaced with the raw hardware access equivalents. This includes replacing the SDL framebuffer code with a `memcpy` to the SDRAM framebuffer, and mapping a select number of buttons to the push buttons on the ML605.

The code base, despite being compact for a desktop application, was quite large for a embedded system on chip. To reliably meeting timing constraints during place and route at a clock speed of 150MHz, the maximum sized BRAM was found to be 256KB. Code size was reduced by re-writing all floating point code to fixed point, which means the binary does not need to be linked to software floating point emulation libraries. While the μblaze is capable of running a floating point unit, this was not used as it affected the ability of the system to meet timing.

The emulator also many statically allocated arrays for holding game data. By placing these in the SDRAM, performance is sacrificed in exchange for the ability to fit the rest of the code in local memory. To reduce the performance impact, a data cache was configured for the SDRAM memory locations where data was placed. This provided a small increase in performance, but reduced the ability of the system to meet timing, so it was discarded.

9.2 Muti-processor Implementation

In order to find more speed, the emulator was spread across multiple tiles. The main instruction set simulator (ISS) was left on the tile with large memories, whilst two more tiles were added: a video tile, and a sound tile.

The video tile was connected to the ISS tile via a FSL with a large FIFO. The FIFO is capable of buffering 16384 words of video data. The data is then fed into a VHDL block which performs the 5:5:5 to 8:8:8 upscaling, and then it is sent down another FSL FIFO to a μblaze which copies the pixels to the framebuffer. The processor is acting as a simple DMA. It is not efficient, as the PLB is capable of receiving 64 or 128 byte transfers, whereas the μblaze is capable of only 32 byte transfers. A PLB DMA unit that Xilinx provides was evaluated for modification to become a FSL capable DMA, however the complexity of modifying the VHDL was considered to too complex to justify the time. Instead a data cache was placed between the μblaze and the SDRAM framebuffer's memory range. When the cache is flushed, it is capable of performing 128 byte writes back to the SDRAM, effectively becoming a DMA for the μblaze.

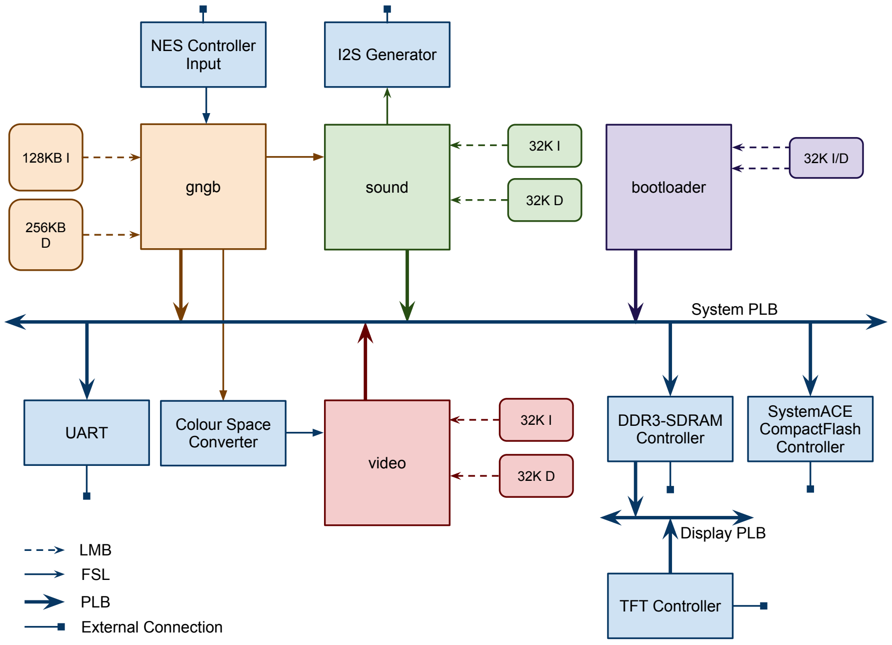

Figure 21: Four Tile GameBoy System

The audio tile required more complex software modifications. The communication with the ISS was simple: in the GameBoy hardware, the audio synthesizer was a ASIC block which was programed via a single register write. This register write became a FSL write from the ISS tile to the audio tile, allowing the audio processing to be completed asynchronously from the ISS. The audio code, as mentioned, has been re-written to perform only fixed point maths. From this is produces a 4 bit sound sample, which is scaled up to a 16 bit sample for the I2S unit receive over FSL and to output to the speakers.

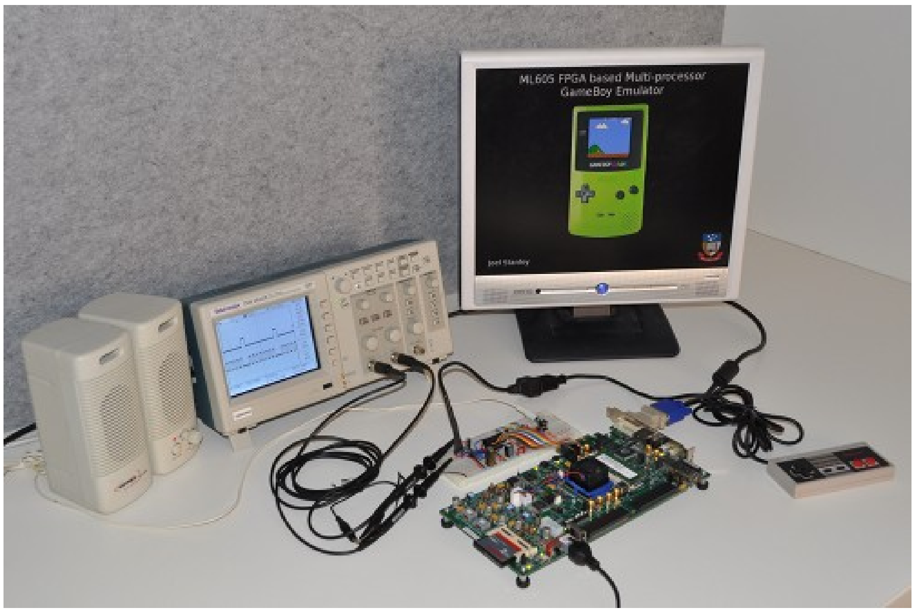

Figure 22: GameBoy System

The resulting system, shown in Figure 22 is able to play GameBoy games with near-real time emulation speeds. This was achievable through considerable software manipulation, but also a well considered selection of hardware interfaces. Many of the transfers are well suited to the message passing buffer that the FSL FIFO provides. As all transfers are uniform in size, a producer can send data to a consumer without having to manipulate any out of band administration as with the memory mapped C-HEAP implementation. The presence of data indicates a full message, and the consumer continues reading a message worth of words.

By relying on full messages to get through, the system is vulnerable to becoming out of sync. It requires the software to be very clear about what is sent over each channel, and thus is not robust. An alternative implementation would be to send a start marker with a length field. The consumer could then discard packets that do not have a start marker, or come after the length field has been exceeded. This still relies on software synchronisation.

10 Summary

10.1 Project Goals

The project outlined three goals in the Design Document at the beginning of the project. These are reproduced in bold, followed by a discussion of the work done in meeting these goals.

Assisting the design of systems with efficient communication characteristics.

The characterisation of the BRAM and SDRAM memories and the latencies that they present when accessed across the LMB and PLB provides a base understanding for producing efficient systems that have a traditional memory-mapped architecture. This was extended through looking at streaming interfaces, the FSL, for applications are memory mapped behaviours were not required.

The production of hardware IP that provides low latency data access, through the avoidance of latency critical communication within systems, without a deep understanding of the native properties of the Microblaze buses.

This project spent considerable time building multi-processor system on chips using design techniques outlined by the surveyed technical papers. By constructing this hardware with the right interconnect for the job, avoiding unnecessary copying of memory through streaming, and always placing the data close to the consumer where possible lead to systems that were affected minimally by latency. This was true to the extent that work was spent speeding up the processing elements - weather that be through more efficient programming, or time spent building IP blocks to accelerate hot spots in code - as the latency issues were optimised as far as the systems would allow.

An area of further exploration that could be perused in the future is building programmable DMAs that are able to accept or provide data over FSL, depending on the application. This was touched on during the project but due to the considerable complexities involved in modifying existing IP, it was not completed.

Enable the team producing the Multi-processor Picture Frame system, as well as future projects undertaken using custom interconnect architectures, to build upon fast, latency-free hardware.

The other team working with the ML605 were provided with hardware where required, and this was accompanied with documentation describing the ML605 IP, and data on how the memories perform under load and in the best case. They were also provided with assistance in the form of engineering expertise when stuck on tooling or conceptual problems, and direction when looking at implementing optimisations. There was unfortunately no reciprocal deliverable of software from the other team, as was verbally agreed upon.

10.2 Project Deliverables

10.2.1 Interconnect Documentation

This deliverable was intended for the other team working with the ML605, as well as future research groups within the school. This was produced as part of the mid year Progress Report, and delivered to the other team shortly after.

The documentation extends beyond the interconnect and peripherals made available by the ML605 to the systems built during the course of this project. This includes a step by step tutorial on how to produce a multi-processor system using Xilinx tools.

10.2.2 MPSoC Communication Architecture

The communication architecture was studied in depth, at both the software layer in the form of message passing, C-HEAP and related abstractions, as well as the hardware layer, through building the multi-processor JPEG and GameBoy systems.

The JPEG system was well engineered for a generic multiprocessor application; it is composable as far as the PLB bus scales and is able to run traditional memory mapped applications, or take advantage of message passing abstractions. In contrast, the GameBoy system was constructed specially for the application - interconnects were in place only where they were required. Devices were not attached to the PLB if they did not require access to the SDRAM, and FSL links were not placed in bidirectional pairs meaning data flow had a predetermined direction.

Whilst it is trivial to add extra interconnect in the GameBoy system, it is noted that the GameBoy is similar to the OMAP3430 discussed in Section 5, whilst the JPEG system is flexible like the Cell BE. It cannot be said that one approach is always better than the other; the GameBoy system would not have met timing at 150MHz if it contained fully connected FSL paths, indicating that the correct approach is dependant on the application.

10.2.3 Demonstration Application

The demonstration application far exceeded initial expectation. By choosing an interactive demonstration, as this attracts those who may not initially have an interest in computer architecture. By adding motion and sound output to the system, latency becomes an externally observable event. Considerable effort was put into ensuring the application can be run independently of a host PC, extending the life of the demo as anyone can now set it up without knowledge of the system. This provides a demonstration for future students, weather they be high school graduates deciding what degree to study, or third years deciding what project to work on.

10.3 Risks Met

There were three important risks that the project had to manage during the project of the 6 that were mentioned in the Design Document.

10.3.1 Data Loss

As detailed in the design document, data loss would be problematic for the project. This occurred on the day before the Progress Report was due, when the laptop that was being used to store data and the report stopped booting due to a broken fan.

Backups were restored and eventually the data was recovered directly from the laptop, leading to a loss of several hours but not permanent data loss.

10.3.2 Inadequate workstation hardware

As this was an expected risk with moderate consequences, it was decided early on that a modern machine would need to be purchased. This occurred within a fortnight, however issues with the networking infrastructure meant the machine was not commissioned until several weeks later. Despite this, the project was able to continue working by using a virtual machine on a laptop to run the project tools.

10.3.3 FPGA not available

Due to the other team not requiring the FPGA until the late stages of the project, there were few moments of contention for the FPGA. Sharing the FPGA with Andreas was a benefit as it provided valuable two-way collaboration.

10.4 Management

As this project only had the one student, there was limited opportunity to employ management skills in the way that most projects would do so. There was considerable effort placed in ensuring progress was documented with daily journal entries. This was monitored through regular contact with supervisors. However the main management aspect of the project was realised through daily interaction with Andreas, the post-doc student who was studying the ML605 until September. Tasks were allocated as the need arose, and progress was reported continually throughout the week.

By working collaboratively on problems, Andreas' knowledge could be employed in areas where he had expertise, and this was reciprocated where my knowledge was greater. By having colleague working in the same office, we were able to employ rubber ducky debugging techniques; explaining the problem out loud in order to identify gaps in understanding, and often this alone would clarify the issue being debugged.

10.5 Significance

11 Conclusion

THIS SECTION UNDER CONSTRUCTION.

11.1 Objectives

This project set out to understand the challenges involved in building a multi-processor system on chip, with a focus on the communication subsystem.

Unique look at ML605 and multiprocessor systems. Design document for this project is the top hit on Google for ML605 multiprocessor.

11.2 Project Progress

The desired objectives for the mid-year point of the project discussed in the Design Document[25] have been met. It was scheduled that a working benchmark system would be built, data collected, and results published, all of which have been achieved on time and below budget. In addition to these goals, there has been the development of a high-resolution TFT controller and the porting of the message passing software API C-HEAP[18] to distributed memory system.

In the process of completing the project goals, a strong familiarity with the Xilinx XPS work flow has been developed, allowing the construction of new test systems to be completed in a number of minutes without assistance. The project also underwent the upgrade to XPS 12.1, bringing official support for the ML605 and limited multi-threaded synthesis capability to the toolchain.

The XPS projects, comprising of both the hardware description and the software, have been maintained in a version control system allowing previous versions to be retrieved and synthesised should the need arise.

Weekly meetings have been held with supervisors, with agendas set and minutes taken. This provides a record of progress, as well as regular checks on progress and advice for external issues such as infrastructure requirements. The minutes complement the project journal which has been updated on a daily basis over the course of the semester.

11.3 Future Work

In addition to exploring software communication models, work will be undertaken to replace the system PLB with the Æthereal Network on Chip[6].

The end deliverable will include a system and software libraries for communication, with the combination providing a capable platform for future MPSoC research.

low-resolution H.263 decoder for picture-in-picture video, cross fading between JPEG images, and MP3 audio decoding.

Comparing C-HEAP to shared messaging

Glossary

APIs: Application Programming Interface

BRAM: Buffered RAM

DDR3 SDRAM: Double Data Rate 3 Synchronous Dynamic Random Access Memory

DMA: Direct Memory Access; a hardware device that performs memory copies

DVI: Digital Video Interface

FPGA: Field-Programmable Gate Array

FSL: Fast Simplex Link

GPIO: General Purpose Input Output

HDL: Hardware Description Language, eg VHDL, Verilog

iDCT: inverse Discrete Cosine Transform

IO: Input Output

JPEG: Joint Photographic Experts Group; a image compression format

LMB: Local Memory Bus

MCU: Minimum Coded Unit

MPMC: Multi-Port Memory Controller

MPSoC: Multi-Processor System on Chip

NUMA: Non-Uniform Memory Architecture

PLB: Processor Local Bus

SDK: Software Development Kit

TFT: Thin Film Transistor

VHDL: Very High Speed Integrated Circuit Hardware Description Language

XPS: Xilinx Platform Studio

ELF: Execute and Link Format; a binary file format

References

[1] Everything you always wanted to know about gameboy, 2001. /url-http://nocash.emubase.de/pandocs.htm.

[2] libusb, 2010. /urlhttp://www.libusb.org.

[3] Inc. Chrontel. Ch7301c dvi transmitter device, 2010.

[4] Cirrus Logic. *Cirrus Logic CS4344/5/6/8 10-pin, 24-Bit, 192 kHz Stereo D/A Converter Data Sheet*, 2005.

[5] Cypress Semiconductor Corporation. Ez-host programmable embedded usb host and peripheral controller, 2008.

[6] Kees Goossens, John Dielissen, and Andrei Radulescu. Æthereal network on chip:concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22:414–421, 2005.

[7] Andreas Hansson. *A Composable and Predictable On-Chip Interconnect*. PhD thesis, Technische Universiteit Eindhoven, 2009.

[8] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):1–24, 2009.

[9] J Helmig. Developing core software technologies for ti's omap platform. 2002.

[10] P. Huerta. A microblaze based multiprocessor soc. volume 4, pages pages 423–430, 2005.

[11] Xilinx Inc. Processor local bus (plb) v4.6 (v1.04a), 2009. /url-http://www.xilinx.com/support/documentation/data_sheets/plbv46.pdf.

[12] Xilinx Inc. Microblaze processsor reference guide edk 12.1, 2010.

[13] Jaume Joven, Oriol Font-Bach, David Castells-Rufas, Ricardo Martinez, Lluis Teres, and Jordi Carrabina. xenoc - an experimental network-on-chip environment for parallel distributed computing on noc-based mpsoc architectures. In *PDP '08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 141–148, Washington, DC, USA, 2008. IEEE Computer Society.

[14] Satya Kiran, M. N. Jayram, Pradeep Rao, and S. K. Nandy. A complexity effective communication model for behavioral modeling of signal processing applications. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 412–415, New York, NY, USA, 2003. ACM.

[15] Alex Krasnov, Andrew Schultz, John Wawrzynek, Greg Gibeling, and Pierre yves Droz. Ramp blue: a message-passing manycore system in fpgas. In *In 2007 International Conference on Field Programmable Logic and Applications, FPL 2007*, pages 27–29, 2007.

[16] Daniel Mattsson. Evaluation of synthesizable cpu cores. Master's thesis, CHALMERS UNIVERSITY OF TECHNOLOGY, 2004.

[17] M. Mehrara, T. Jablin, D. Upton, D. August, K. Hazelwood, and S. Mahlke. Multicore compilation strategies and challenges. *Signal Processing Magazine, IEEE*, 26(6):55 –63, nov. 2009.

[18] André Nieuwland, Jeffrey Kang, Om Prakash Gangwal, Ramanathan Sethuraman, Ramanathan. Sethuraman@philips. Com) Natalino Busa, Kees Goossens, and Rafael Peset Llopis. C-heap: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems, 2002.

[19] University of Eindhoven. 5kk03 - embedded systems laboratory, 2010. /url-http://www.es.ele.tue.nl/education/5kk03/index.shtml.

[20] Pierre G. Paulin, Chuck Pilkington, Michel Langevin, Essaid Bensoudane, and Gabriela Nicolescu. Parallel programming models for a multi-processor soc platform applied to high-speed traffic management. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 48–53, New York, NY, USA, 2004. ACM.

[21] D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *Solid-State Circuits, IEEE Journal of*, 41(1):179 – 196, jan. 2006.

[22] Nathaniel Pinckney, Thomas Barr, Michael Dayringer, Matthew McKnett, Nan Jiang, Carl Nygaard, David Money Harris, Joel Stanley, and Braden Phillips. A mips r2000 implementation. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 102–107, New York, NY, USA, 2008. ACM.

[23] Charlie Ross. Using fifos in hardware-software co-design for fpga based embedded systems. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 318–319, Washington, DC, USA, 2004. IEEE Computer Society.

[24] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[25] Joel Stanley. Multi-processor system on chip communication acceleration architecture design document. 2010.

[26] Antonino Tumeo. A design kit for a fully working shared memory multiprocessor on fpga. In *GLSVLSI '07: Proceedings of the 17th ACM Great Lakes symposium on VLSI*, pages 219–222, New York, NY, USA, 2007. ACM.

[27] S. van Haastregt and B. Kienhuis. Automated synthesis of streaming c applications to process networks in hardware. pages 890–893, apr. 2009.