Lund University

Department of Electrical and Information Technology

Master of Science Thesis

AMBA AXI Video Display Controller supporting paged memory

Author:
Emad Malekzadeh Arasteh

Supervisor:
Ola Hugosson
Examiner:
Joachim Rodrigues

Lund 2011

©

The Department of Electrical and Information Technology
Lund University
Box 118, S-221 00 LUND
SWEDEN

This thesis is set in Computer Modern 10pt,
with the LaTeX Documentation System

© Emad Malekzadeh Arasteh 2011

Abstract

This Master's Thesis addresses the development of a high performance HD video display controller supporting paged memory. A study of the AMBA protocol, the de-facto interconnect standard, is carried out followed by an investigation of handling virtual memory both in ARMv7 and Linux. The architecture coping with high system memory latency and targeting low power and area has been presented. Implementation features, verification process and synthesis result are described in detail. Lastly, the appendix provides some design related video basics employed in the course of the project.

Contents

1 Introduction **6**
1.1 Overview . . . 7
1.2 Features . . . 7

2 SoC Interconnect **9**
2.1 System-on-Chip communication . . . 10
2.2 Bus-based communication . . . 10
2.3 AMBA . . . 11

3 Virtual Memory Management **14**
3.1 Virtual Memory . . . 15
3.2 ARMv7 VMSA . . . 17
3.3 Paging in Linux . . . 21
3.4 ARMv7 Linux . . . 22

4 System Architecture and Hardware Implementation **24**
4.1 Requirements and Features . . . 25
4.2 Implementation . . . 30
4.3 Verification . . . 37
4.4 Synthesis . . . 38

Conclusion **40**

A Appendix **41**
A.1 Y'CbCr color space . . . 41
A.2 HDMI standard . . . 42

List of Acronyms

AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
AXI	Advanced eXensible Interface
FIFO	First In, First Out
FPS	Frame Per Second
HD	High Definition
HDMI	High-Definition Multimedia Interface
IRQ	Interrupt Request
LCD	Liquid Crystal Display
PTE	Page Table Entry
RGB	Red Green Blue
UMA	Unified Memory Architecture
VDC	Video Display Controller
VE	Video Engine
VGA	Video Graphics Array
VMSA	Virtual Memory System Architecture
YUV	Linear transform color space
Y'CbCr	Gamma corrected color space

Chapter 1

Introduction

In todays consumer electronic market, the public has come to expect HD video displays with all of their electronic devices from phones to PDAs to gaming consoles. These video displays, which are directly exposed to the end-user and are a center of focus, play a significant role in the quality of a user's experience.

The Video Display Controller (VDCs), as a final piece in the chain of numerous components in a multimedia device, carries a considerable portion of the burden in producing a high quality experience. SoC designers have the challenge of producing both high-performance and low-power video display controllers.

1.1 Overview

The Video Display Controller (VDC) generates video signals for driving an LCD or a VGA screen. It's the VDC's responsibility to fetch the video data and maintain the timing of necessary video signals including horizontal and vertical synchronization signals, and the blanking interval signal.

The VDC must flawlessly perform its functionality in spite of high latency in the system. Torn or cluttered video frames certainly are not pleasant. Video FIFO queues store the pixel values inside the SRAM decoupling the system memory from the VDC. A simplified system including the VDC is shown in figure 1.1.

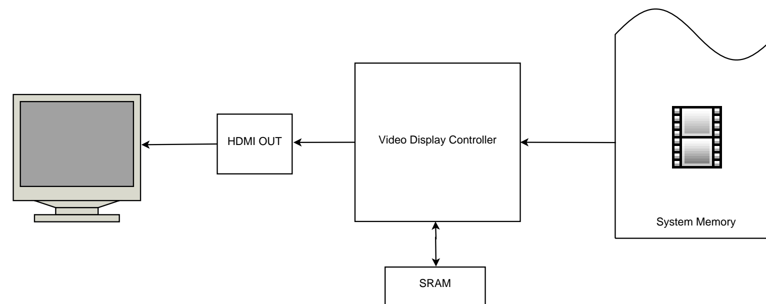

Figure 1.1: Simplified system including VDC

1.2 Features

When designing a VDC for an embedded environment, several constraints arise which are not present for desktop applications. The VDC presented in this document includes the following features:

- Supporting 300 clock cycles average bus latency
- Supporting 10,000 clock cycles peak bus latency (without dropping pixels)
- Virtual memory paging, 4 KiB page size
- 1-level page table stored in the external RAM
- Pixel FIFO queue implemented with single ported SRAM
- AMBA AXI Master 64-bit DMA supporting outstanding and out-of-order transactions
- AMBA APB Slave Register File

- Support for two layers (picture-in-picture) with RGBα (1 plane) and YUV444 (3 planes) as input formats
- Configurable stride for all buffers
- Support for non-paged aligned buffer accesses
- Insensitive to CPU IRQ latency
- Customizable timing signal generation

Video resolution for ASIC and FPGA assumes Full HD and VGA respectively with 60 FPS frame rate and 60 Hz screen refresh rate.

This document begins with describing the importance of bus architecture in today's SoC design in Chapter 2 and follows by introducing the AMBA bus protocol as applied interconnect standard in the VDC implementation. Chapter 3 concentrates on virtual memory architecture from both hardware and software perspectives. ARMv7 Virtual Memory System Architecture (VMSA) and Linux Paging are investigated as case studies to approach paged memory VDC design. Chapter 4 focuses on VDC system requirements and hardware implementation. Finally, test environment for verification and synthesis results are presented.

Chapter 2

SoC Interconnect

Everyday emergence of variant complex applications demands faster traverse of data on a chip. Meanwhile design productivity gap compels adoption of reusable Intellectual Properties (IPs) to reduce design time for integration. These leads designers toward high performance and low-power interconnect architecture for ensuring reliable on-chip communication.

2.1 System-on-Chip communication

System-on-Chip (SoC) interconnect alternatives are bus, switch and Network-on-Chip. Choosing a suitable interconnect architecture demands a good understanding of two systems level concepts:

1. **Communication bandwidth** rate of information transferred between module and the surrounding environment in which it operates and usually measured in bytes/second.

2. **Communication latency** time delay between a module requesting data and receiving response to the request[1].

On-chip interconnect architecture must guarantee demanded bandwidth and latency in order to satisfy application constraints. Albeit, selection and implementation of interconnect architecture ensuring application performance must not violate overall chip area and power by itself. Therefore, designing on-chip communication architectures has become a serious challenge seeking a careful and time-consuming decision process[2]. Due to the VDC design environment, this chapter mainly focuses on the bus-based communication.

2.2 Bus-based communication

Busses are the most widely means of on-chip communication between components in SoC design. The simplicity and efficiency of busses made them the preferred interconnect architecture for today SoC designers[2].

Bus in concept is just shared wires between components and in practice some logic is added to make orderly usage of the bus. *Master* unit initiates communication on bus which *slave* unit responds to a request. For avoiding conflict between different components in simultaneous usage *arbitration* process determines ownership of the bus. The arbitration unit grants bus ownership to one requesting as determined by *bus standard*[1].

Since the early 1990s, several on-chip bus-based communication architecture standards have been proposed to handle the communication needs of emerging SoC designs. Some of the popular standards in industry are Advanced Microcontroller Bus Architecture (AMBA) developed by ARM Ltd, CoreConnect developed by IBM and OpenCores Wishbone as an open source standard.

2.3 AMBA

Advanced Microcontroller Bus Architecture(AMBA) 2.0 was introduced in 1997 and defines an on-chip communication standard for designing high-performance embedded microcontrollers. Three distinct buses are defined within the AMBA 2.0 specification:

- the Advanced High-performance Bus (AHB)
- the Advanced System Bus (ASB)
- the Advanced Peripheral Bus (APB)

The AMBA specification has been derived to satisfy four key requirements[3]:

- Facilitate right-first-time development
- Technology-independent and highly reusable
- Modular system design
- Minimize the silicon infrastructure

Later AMBA 3.0 introduces the Advanced eXensible Interface (AXI) bus that extends AHB with higher performance features.

An AMBA-based microcontroller typically consists of a high-performance system backbone bus (AMBA AXI), able to sustain the external memory bandwidth, on which the CPU, on-chip memory and other Direct Memory Access (DMA) devices reside. This bus provides a high-bandwidth interface between the elements that are involved in the majority of transfers. Also located on the high performance bus is a bridge to the lower bandwidth APB, where most of the peripheral devices in the system are located (Figure 2.1).

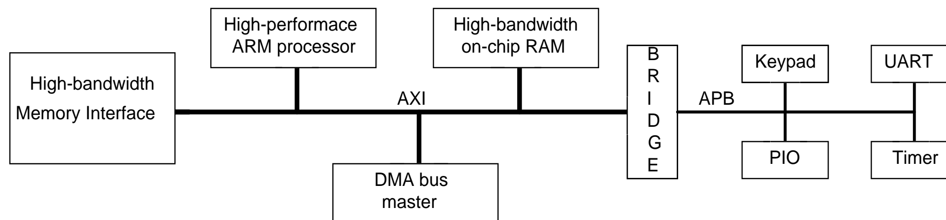

Figure 2.1: Typical AMBA-AXI based system[3]

In the video display controller design, AXI is used to connect shared external memory, ARM CPU and VDC. Control registers in the VDC are accessed by APB

interface. Therefore, a quick review of the AXI and the APB features are carried out in two following subsections.

Advanced eXensible Interface (AXI)

The objectives of the latest generation AMBA interface include supporting high-bandwidth and low-latency designs without using complex bridges, being suitable for memory controller with high initial access latency, adaptability in meeting interface requirements and being backward-compatible with existing AHB and APB interfaces[4].

AXI is a burst-based standard proposing pipelined data transfer. It describes a channel based architecture for communicating between masters and slaves on a bus. Five separated channel are defined: **read address**, **read data**, **write address**, **write data**, and **write response**. Figure 2.2 indicates a read transaction uses the read address and read data channel.

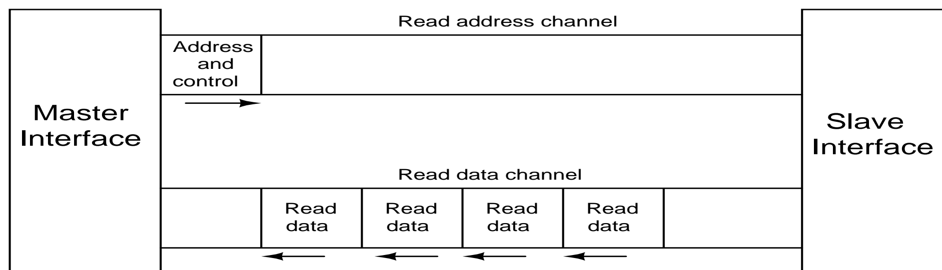

Figure 2.2: AXI read transaction[4]

Key feature of the AXI protocol which is widely exploited in the current VDC design is ability to issue multiple outstanding addresses means that master can issue transactions addresses without waiting for earlier transaction to complete. This feature makes maximum channel utilization by enabling parallel processing of transactions.

Another interesting feature of AXI buses is the ability to complete transactions out of order. It simply means those requests to faster memory regions are performed earlier than requests to slower memory regions.

Above mentioned features makes AXI standard a perfect choice for the VDC implementation which has latency critical and bandwidth sensitive requirements.

Advanced Peripheral Bus (APB)

The APB bus protocol is optimized for minimal power consumptions and reduced interface complexity. The AMBA APB should be used to interface to any peripherals which are low-bandwidth and do not require the high performance of a pipelined bus interface. The APB allows only non-pipelined data transfers, and has only a single master. The state diagram, shown in Figure 2.3, represents transfer activity of APB bus. **IDLE** is default state for the peripheral bus which moves to **SETUP** state by receiving transfer request and asserting the appropriate slave select signal to participate in the transfer. The bus only remains one clock cycle in **SETUP** state and always move to **ENABLE** state on the next rising edge of the clock. In **ENABLE** state the enable signal, **PENABLE** signal is asserted to indicate that the transfer is ready to be performed. The **ENABLE** state also only lasts for a single clock cycle and either will return to **IDLE** state or **SETUP** state based on existence of any further transfer request.

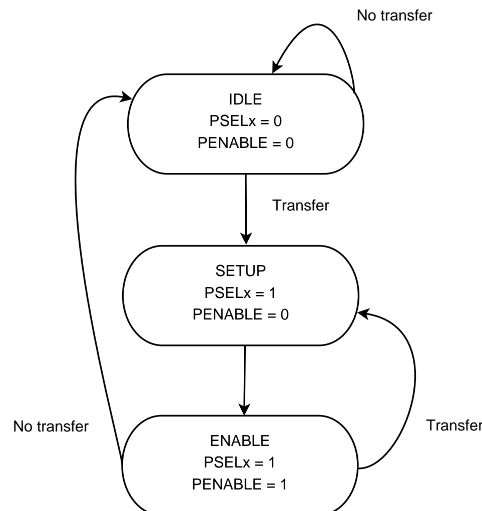

Figure 2.3: APB bus state diagram[3]

Chapter 3

Virtual Memory Management

Today SoCs are made of highly integrated software and hardware components. In order to achieve optimal system partitioning, it's essential to gain an understanding of system and software alongside by hardware and circuit. Virtual memory management by having its root in software is a 'fact' in today embedded applications. Therefore, VDC design won't realize without good insights of handling virtual memory in both CPU and operating system.

3.1 Virtual Memory

Virtual memory is a memory management technique that realizes existence of multiple processes at any instant in time. Each process has its own address space in memory and dedicating a full address space to every process is too expensive. Virtual memory suggests to divide physical memory into blocks and allocate them to different processes. Therefore, process has the illusion of exploiting a full contiguous address space but in reality some of its active parts are scattered around memory and the inactive parts are saved in a disk file. System supporting virtual memory inherently enjoys numerous advantages including expendability, relocation and protection of memory[5].

First, we have to distinguish two types of addresses used in systems adopting virtual memory:

virtual address refers to address used by operating system to access kernel and user memory.

physical address used by memory controller to address cells inside physical memory.

The processor produces virtual addresses that by a combination of operating system and Memory Management Unit (MMU) are translated to physical addresses, which access main memory.

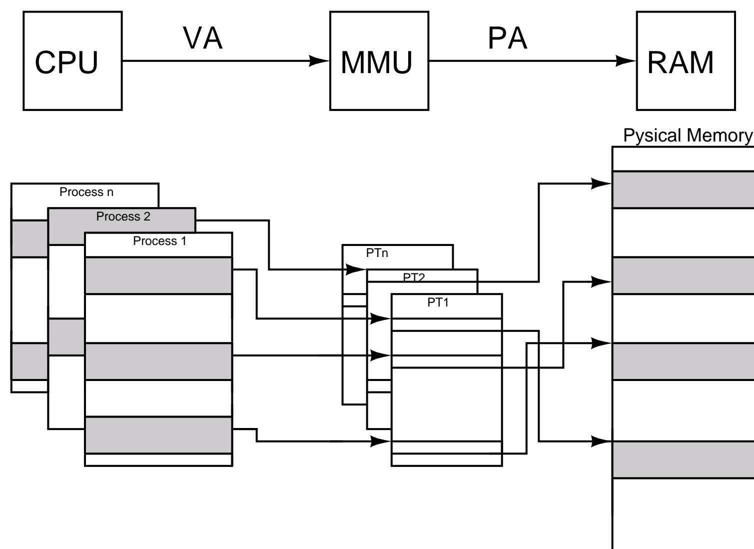

Figure 3.1: Virtual to physical address translation

Paging

Almost all implementations of virtual memory use **paging** scheme. Paging permits the physical address space assigned to each process to be noncontiguous. The basic method for implementing paging involves breaking physical address space into fixed-size blocks called **frames** and breaking virtual address space into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from the backup storage[6].

Physical base address of each page is stored in a data structure called **page table**. When the CPU generates a virtual address, the address divided into two parts a **page number(p)** and a **page offset(d)**. The page number is used as an index into a page table. Then, the page offset is simply concatenated to physical page base address that is sent to the memory unit. With this scheme, *two* memory accesses are needed to access a word in memory (one for the page base address, one for the word). To reduce address translation time, MMU uses a fast lookup dedicated cache namely **translation look-aside buffer (TLB)**. A TLB entry contains a tag part which holds portion of virtual address and the data holds a physical page frame number, protection field, valid bit, and usually a use bit and dirty bit. If desired page number is not in the TLB (**TLB miss**), a memory reference to the page table must be made. Figure 3.2 illustrates virtual to physical address translation using TLB.

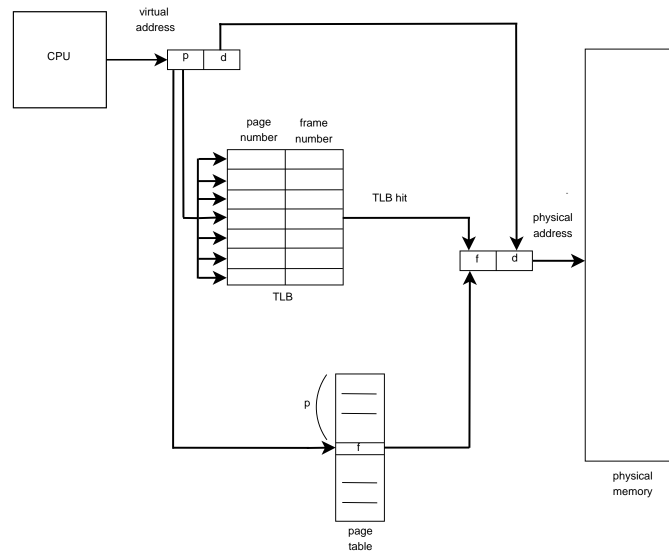

Figure 3.2: Paging hardware with TLB[6]

The page size is defined by the hardware. Given a 32-bit virtual address, 4 KiB pages, and 4 bytes per **page table entry (PTE)**, the size of the page table would be $(2^{32}/2^{12})* 2^2= 2^{22}$ or 4 MiB. Obviously, it's not desirable to allocate the page table contiguously in main memory. One solution is to page the page table itself that implied **two-level page table** scheme. Since we paged the page table, virtual address is further divided into another part called **second-level table**. Figure 3.3 shows two-level page table address translation where p_1 is an index into the second-level table, p_2 is an index to the page table and d is an offset inside the page.

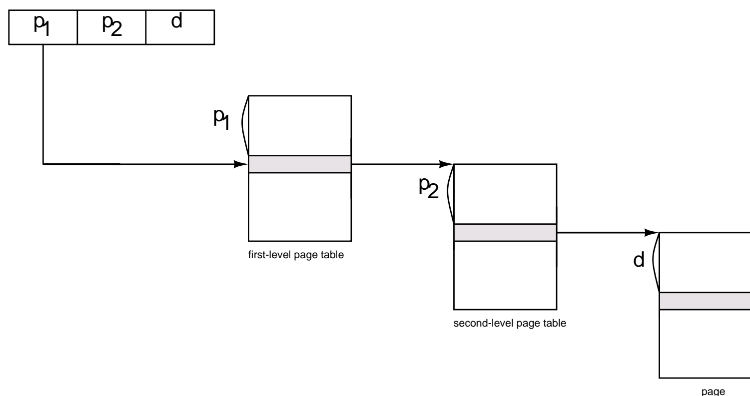

Figure 3.3: Two-level page table[6]

Using previous example, virtual address was divided into a page number (20 bits) and a page offset (12 bits). Now by assuming 12 and 8 bits long for second-level and first-level table (page table) field, they can include up to 4096 and 256 entries. It means second-level table can address $4096*256*4096=2^{32}$, as expected in 32-bit address architecture.

3.2 ARMv7 VMSA

ARMv7 VMSA is referred to Virtual Memory System Architecture of ARM Cortex™ processor family. In a VMSA, MMU provides facilities for an operating system to dynamically allocate memory and other memory-mapped system resources to the processes. The MMU handles virtual to physical address translation by holding associated memory properties in memory-mapped tables known as translation tables.

Translation tables

Since ARM CPUs are used in wide embedded applications with variable memory requirements, the MMU supports memory accesses based on sections or pages. Su-

persections and sections consists 16 MiB and 1 MiB blocks of memory. Large and small pages consists 64 KiB and 4 KiB blocks of memory. Supporting Supersections, Sections and Large pages permits mapping of a large region of memory by just using a single TLB entry.

Translation tables held in memory have two levels :

- **First-level tables** holds first-level descriptors that contain either the base address for Sections and Suppersections or pointer to second level table for Large pages and Small pages
- **Second-level tables** (*page tables*) holds second-level descriptors that contain the base address of a Small page or a Large page

Figure 3.4 shows how first-level and second-level table assist to translate a virtual address in a case of Small page. Worth mentioning, descriptors in addition to base addresses contains translation properties. These control fields identify descriptor type, memory region attribute, access permission etc. Explaining each level descriptor is a tedious process which would not expressed here.

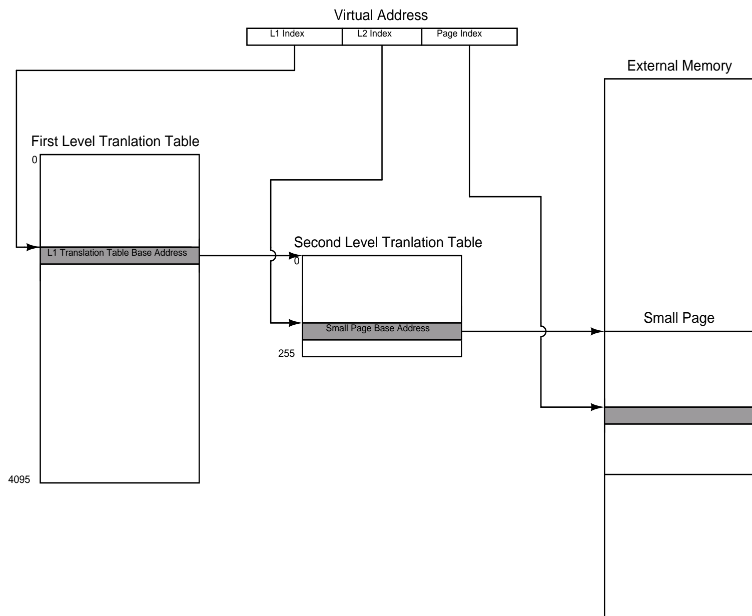

Figure 3.4: ARM Cortex address translation

Translation Table support registers

When the VMSA is implemented, three translation table support registers are used: TTBR0 and TTBR1 holds the base address of translation table 0 and 1, respec-

Table 3.1: VMSA translation table support register

Register name	Description
Translation Table Base 0	c2, *Translation Table Base Register 0 (TTBR0)*
Translation Table Base 1	c2, *Translation Table Base Register 1 (TTBR1)*
Translation Table Base Control	c2, *Translation Table Base Control Register (TTBCR)*

tively. When TLB miss happens, the Translation Table Base Control Register, TTBCR, determines which of Translation Table Base Registers defines the base address for the translation table walk.

Access to translation table support registers are only possible in privileged mode. For accessing e.x.TTBR0, one should read and write the CP15 registers with <opc1> set to 0, <CRn> set to c2, <CRm> set to c0, and <opc2> set to 0.

```
MRC p15,0,<Rt>,c2,c0,0 ; Read CP15 Translation Table Base Register 0
MCR p15,0,<Rt>,c2,c0,0 ; Write CP15 Translation Table Base Register 0
```

Virtual to physical address translation flow for a Small page is shown in figure 3.5.

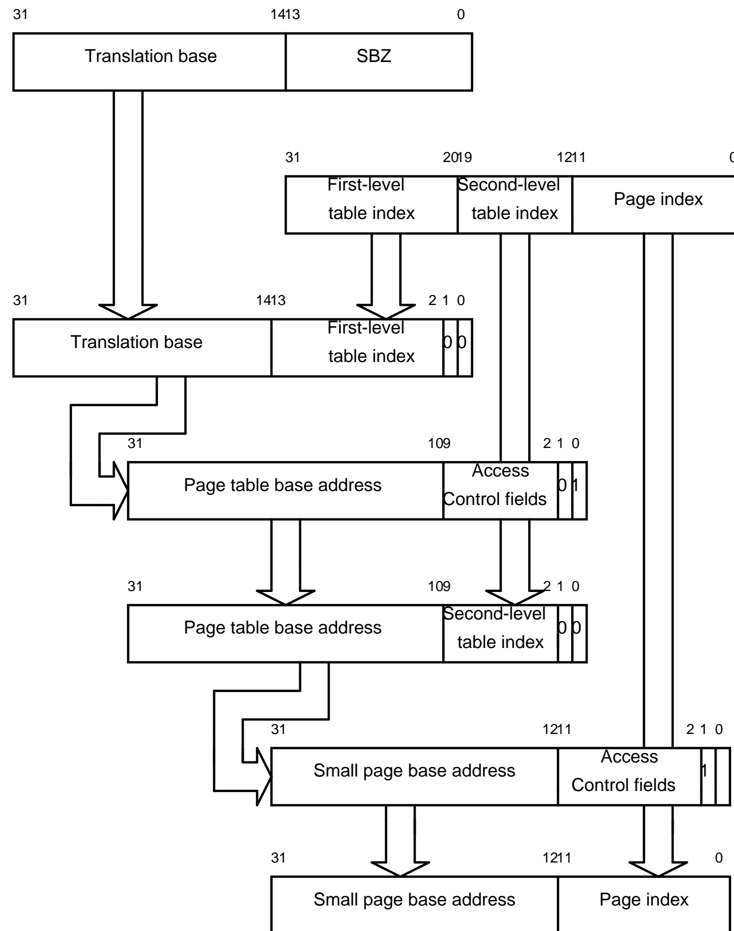

Figure 3.5: Small page address translation flow[7]

3.3 Paging in Linux

Linux always tries to keep a neat distinction between hardware-dependent and hardware-independent source codes. Therefore, the Linux Memory Management scheme adopts a common paging model to fit different architectures. Paging model in Linux consisted of there paging level :

- Page Global Directory (PGD)
- Page Middle Directory (PMD)
- Page Table Entry (PTE)

Page Global Directory includes the addresses of several Page Middle Directories, which in turn includes the addresses of several Page Tables. Finally, each Page Table entry points to a actual physical page frame in main memory. Address translation in Linux virtual memory containing 3 paging level is depicted in Figure 3.6.

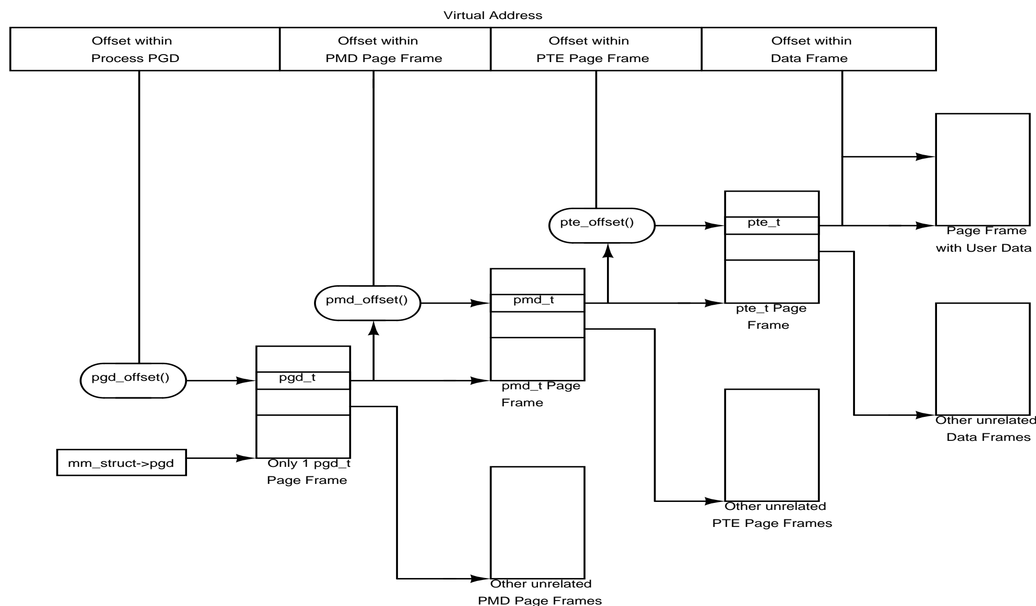

Figure 3.6: Linux page table layout[8]

All related information to the process address space is included in an object called the *memory descriptor* of type mm_struct [9]. (mm_struct–>pgd) points to base address of Page Global Directory which is a physical page frame. This frame contains an array of type pgd_t pointing to a page frame containing an array of Page Middle Directory entries of type pmd_t which in turn points to page frame containing Page Table entries of type pte_t, which finally points to page frames containing the actual user data[8].

3.4 ARMv7 Linux

ARMv7 has two level page table structure, where the first-level has 4096 entries and the second-level has 256 entries. Each entry is one 32-bit word. But as mentioned, Linux has 3 level page table structure which must be tweaked slightly to accommodate the two-level scheme.[7]

First, size of PMD needs to be changed to one. It means Middle Directory table would only have 1 entry and results in bypassing PMD. Since "all references to an extra level of indirection are optimized away at compile time not at run time, there is no performance overhead for using generic three-level design on platforms which support only two levels in hardware"[10].

Second, sizes of PGD and PTE require to be defined following Linux memory management codes' restrictions. Linux keeps two sets of PTEs- the hardware and the Linux version. During translation table walks, MMU just reads hardware PTEs and operating system uses Linux PTEs to check flags such as present and dirty bits to handle page tables in optimum way. Linux also expects one PTE table per page.

Eventually, for adopting an ARM page table, number of entries in Linux first page level must be set to 2048 with the size of 8 bytes for each entry (whereas ARM first level page table has 4096 entries of 4 bytes each) and 512 entries in the second level page table (whereas ARM second level page table has 256 entries). It means every entry in the first level table contains two second level pointers and the second level table contains two hardware PTE tables arranged contiguously. Second level table in addition to two hardware PTE contains two Linux PTE tables[11]. This leads to the following page table layout in ARM Linux :

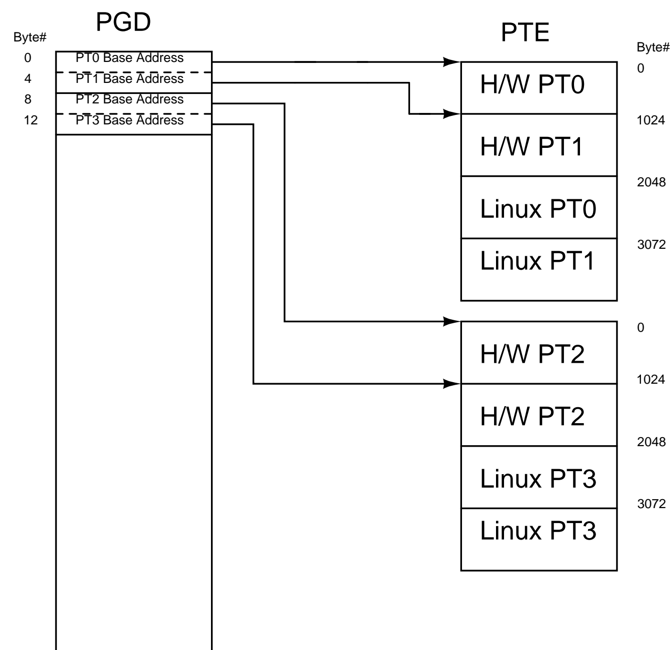

Figure 3.7: ARM Linux page table layout

Chapter 4

System Architecture and Hardware Implementation

While PC users partially tolerate dropped frames and stuttering, for users of quality consumer products such as smartphones and tablets inconsistent video frame rates are definitely unpleasant. Moreover, minimizing area and power are never ending challenges for todays SoC designers to ensure a high quality consumer product at a reasonable price.

4.1 Requirements and Features

Precise investigation of the system constraints results in a neat specification which by itself fastens the implementation process. This section describes design requirements in detail that alleviates the course of the VDC development.

UMA vs video RAM

Classic PC architecture proposes separate video RAM known as a *frame buffer* for graphic and video applications. But having additional sub-system memory is very costly in handheld devices. Moreover, it demands complex memory management scheme due to two different kinds of memory in the system and also results in memory that is not easy to use because it's dedicated to another purpose. Therefore, it's preferred to use Unified Memory Architecture (UMA) in multi-function devices like smartphones with highly variable workload. Figure 4.1 depicts a typical mobile platform containing CPU, GPU, Video Engine (VE) and VDC sharing single memory. Current approach enjoys lower cost and greater flexibility.[12]

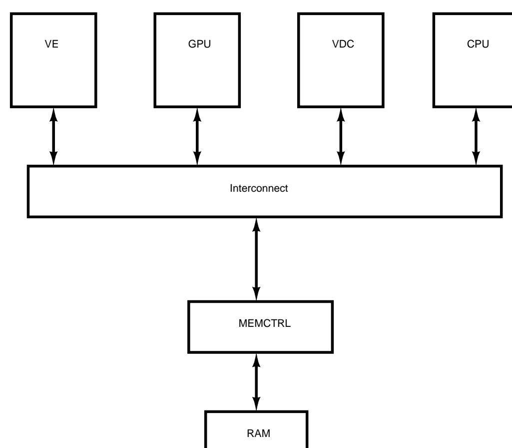

Figure 4.1: Unified Memory Architecture (UMA)

As discussed in previous chapter, virtual memory technique is a fact in all today system architectures. CPU manages memory using paging scheme and makes it impossible to guarantee a contiguous physical memory allocation from user space. By adopting UMA, all the modules connected to system shared memory including VDC must maintain their own address translation unit in order to access virtual memory.

Video frame allocation in Virtual Memory

Single video frame on the screen is blend of multiple *layers*. Layers can be in various formats e.g. YUV444, YUV422, YUV420, RGB or RGBα. Color components in each layer may be packed in a single array resulting an interleaved version or each stored in a separate array resulting several *planes*. A typical situation contains two layers: YUV format as a video layer and RGBα format as a user interface layer. (See Appendix regarding details about YUV and RGB color space)

Applying virtual memory, a frame is allocated in a noncontiguous memory space. Therefore, a frame and in turn its layers are paged and scattered around the physical memory. The Page List keeps base addresses of physical pages of a video frame inside an external memory. Depending on a layer format, each color component inside a frame can have its own Buffer Page List (BPL) or shares a single BPL. But layers always have a different BPLs and are not allowed to share one together.

For a use case of two separate layers containing YUV444 stored in 3 different planes and RGBα stored in a single plane, 3 BPLs keep physical base addresses of Y, U and V color components and only one single BPL keeps RGBα ones. The BPL is described as a 1-level translation table containing 32-bit base addresses of 4 KiB buffer physical pages. The BPL by itself stores in a continuous physical memory inside an external memory and can not be paged again. The BPL entries are stored in a 64-bit aligned format by appending 32-bit zeros to their most significant sides.

The Buffer Size and the Buffer Page List Size (BPLS) are determined by Formula 4.1 and Formula 4.2 respectively:

$$Buffer\ size = (Unit\ Size \times Buffer\ Stride \times Buffer\ Height) + Original\ Page\ Offset \tag{4.1}$$

$$BPLS = \begin{cases} (Buffer\ Size)/4096 & \text{Buffer Size} \equiv 0 \quad (4096) \\ (Buffer\ Size)/4096 + 1 & \text{Buffer Size} \not\equiv 0 \quad (4096) \end{cases} \tag{4.2}$$

Where Buffer Height- is a 16-bit value indicating number of pixels in vertical direction for the entire buffer; Buffer Stride (BS)- is also 16-bit value containing the difference in bytes between vertically adjacent pixels; Original Page Offset is a 64-bit aligned 12-bit value allowing non-paged aligned buffer accesses by defining an offset value to a base address of the first page inside a buffer and Unit Size is a number of bytes per pixel inside a buffer.

The Buffer Page List Address (BPLA) is a 32-bit physical address indicates address of the first entry inside the BPL. And as previously mentioned, depending on the layer format each layer may have several BPLs and consequently several BPLAs. Since the BPLs of each layer are stored in a continuous memory space, BPLAs are computed depending on each Buffer Size and during run time.

To conclude, every BPL is defined by three parameters: BPLA, BPLS and BS. Figure 4.2 shows simple placement of YUV layer using 1-level page table scheme in the external memory for the mentioned use case.

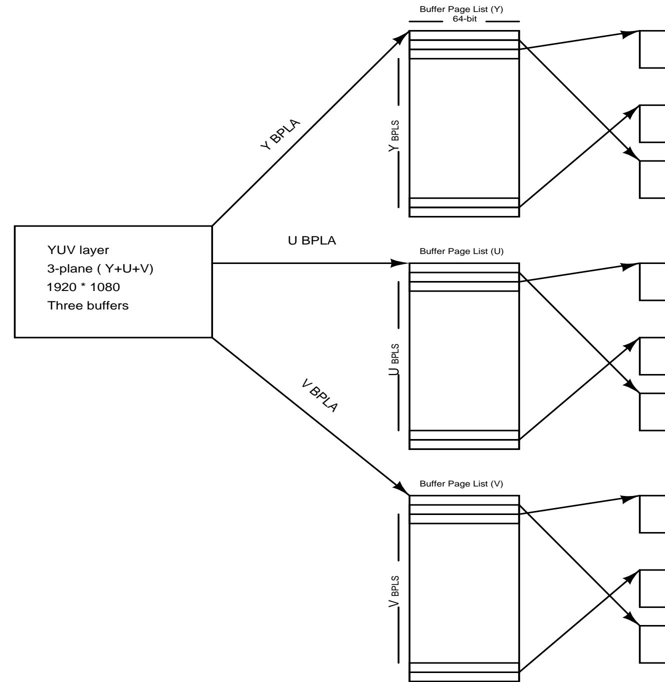

Figure 4.2: YUV layer page lists

Latency-critical and Bandwidth-sensitive VDC

As mentioned in Chapter 2, latency and bandwidth are two fundamental requirements of every master connected to an interconnect. These requirements vary enormously by master type. For example, CPU performance is highly dependent on effective latency of memory since normally CPU can not process further until memory access is completed. In contrast, GPU and VE requires very high memory bandwidth because they typically "performs the same or very similar operations on huge numbers of individual data elements"[12].

The VDC must output pixels at a constant rate. Underflowing frame FIFO queues would break up the image on the screen which is very unpleasant. Thus functionality of the VDC is critically dependent on the bus latency meanwhile its bandwidth

requirement is moderate in comparison with GPU or even VE.

Bus latency emerges in two forms: individual peak latency and long-term average latency. Peak latency mitigates by introducing FIFO buffers inside SRAM. Average latency impact fades by supporting outstanding transactions and increasing burst length. Formula 4.3 indicates the relation between bus bandwidth and required VDC bandwidth where BW_{BUS} indicates nominal bus bandwidth and Outstanding Credit is number of outstanding transactions that VDC is allowed to issue.

$$BW_{VDC} = \frac{BW_{BUS}}{Mlat_{avg}} \times (BurstLength) \times (OutstandingCredit) \tag{4.3}$$

For a use case of 1080p60 supporting two layers containing YUV444 and RGBα and assuming 8-bit resolution on input, we would need 7 bytes for showing each pixel resulting a required bandwidth for the VDC:

$$BW_{VDC} = 1920 \times 1080 \times 60 \times 7 \simeq 871\ MB/s. \tag{4.4}$$

Having 64-bit data bus fed by 300 MHz clock frequency makes nominal bus bandwidth of 2400 MB/s. Using 4.3 and supporting 300 clock cycle average latency in maximum possible burst length (16 beats) case led to the essence of assigning 7 outstanding burst credit for the VDC (Formula 4.5).

$$871M = \frac{2400M}{300} \times 16 \times (OutstandingCredit) \Rightarrow OutstandingCredit \simeq 7 \tag{4.5}$$

Worth mentioning, the average bus latency hit the required bandwidth and made the VDC a bandwidth-sensitive master. But as far as the minimum required bandwidth is met, improving it has no performance impact.

As previously discussed in the case of facing bus peak latency of 10,000 clock cycles, the FIFO buffers inside the SRAM will continue to feed the screen. Size of the FIFO buffers are calculated using Formula 4.6:

$$FIFO\ Size = T_{BusOff} \times BW_{VDC} \tag{4.6}$$

T_{BusOff} is a period in time that bus are not able to deliver the VDC any data that obtained by 4.7:

$$T_{BusOff} = \frac{Peak\ Bus\ Latency}{Bus\ Frequency} \tag{4.7}$$

Refereing to the previous use case of 1080p60 supporting two layers, the FIFO sizes for each separate buffer of RGBα, Y,U and V would be :

$$T_{BusOff} = \tfrac{10000}{300M} = 33\mu S \tag{4.8}$$

$$FIFO_{RGB\alpha} = 33\mu * 125M * 4 \simeq 16KiB \tag{4.9}$$

$$FIFO_Y = FIFO_U = FIFO_V = 33\mu * 125M * 1 \simeq 4KiB \tag{4.10}$$

Dual-ported RAM vs double single-ported RAM

Use of dual-ported RAM is quite common case in Video RAM (VRAM) for desktop applications. It allows the CPU to draw the frame at the same time at the VDC reading it to the screen. Basic dual port RAM cell contains 8 transistors while single port RAM has 6 transistors. In low power environment like handheld devices that every single transistor counts using dual-ported RAM is quite expensive.

To minimize power and area alongside by maintaining required bandwidth, address space for referring pixel FIFO queues are mapped to two single port memories using least significant address bit (LSB) of FIFO pointers. This makes a zigzag memory access pattern allowing almost simultaneous write from the system memory and read by the VDC.

Double-buffering (Double-Job)

Flicker and tearing are undesired artifacts that appear in a case of using a single frame buffer. *Double-buffering* technique avoids incomplete update of a frame buffer by taking advantage of two buffers namely *back buffer* and *front buffer*. While the host program is writing in the back buffer, the VDC is showing the front buffer on the screen. Interrupt request by the VDC informs the host CPU to fill the back buffer and update the VDC register files.

The term of 'buffer' is widely used to refer elements inside a layer in this thesis context. Therefore, the term 'job' is alternatively employed to describe an entire frame consisting of its layers and in turn its buffers. Figure 4.3 illustrates the hierarchy to handle a sequence of video frames for the mentioned use case of two layers.

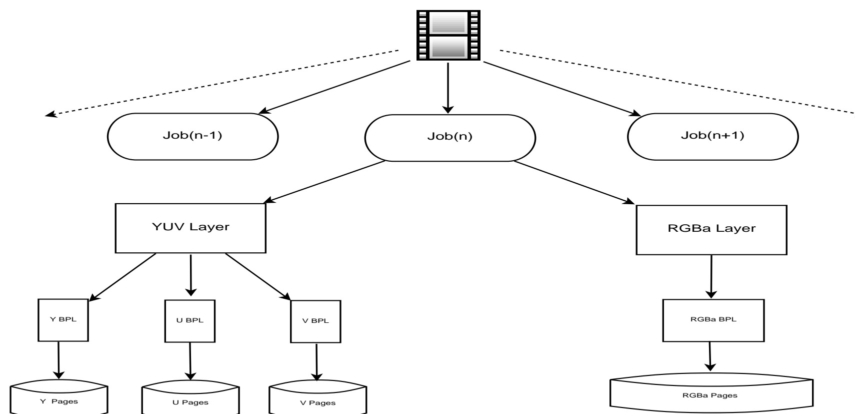

Figure 4.3: The VDC hierarchy to handle a frame sequence

Interrupt handling is implemented by *poll* file operation inside the VDC character device driver and a read/write register inside the hardware. In the case of having two layers, sending an interrupt is postponed until all the buffers for the front job complete issue of their pixel addresses. Figure 4.4 shows Interrupt Request Register Mode for handling double jobs.

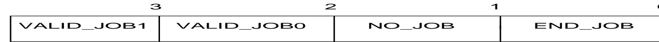

Figure 4.4: IRQ MODE Register

END_JOB sets by the VDC to signal the host CPU about an interrupt and interrupt service routine clears the bit once handles the interrupt request. NO_JOB defines the requested interrupt belongs to a front job or a back job. Since it takes some time from handling service routine until the back job filled, VALID_JOB bit assures availability of the entire frame to the VDC. In case of unavailability of the new frame, the VDC decides to show the previous frame one more time until back job updates its content.

4.2 Implementation

VDC consists of three main parts: Direct Memory Access (DMA) unit, Display Formatter (DFMT) unit and Host interface. A simplified step-by-step functional flow assists a plain understanding of the system process and usage of each unit:

1. Host CPU programs the VDC control registers including display resolution, display position and buffer definitions.
2. DMA translates virtual addresses to physical ones and fetch video frame data from system shared memory and stores them in video FIFO queues inside SRAM.
3. DMA always assures adequate pixels in FIFO queues to avoiding underflow condition due to high peak latency.
4. DFMT retrieves pixel values from SRAM and forms the output signal to be displayed on screen.
5. Host interface signals the CPU once the front buffer is shown and asks for loading the back buffer.
6. If host CPU has written the new frame, VDC updates its control registers; otherwise it keeps the registers values and decides to show the previous frame one more time.

7. Goes to step 2.

Figure 4.5 shows top-level block diagram of the VDC supporting 2 layers-YUV444 (3-planes) and RGBα (1-plane).

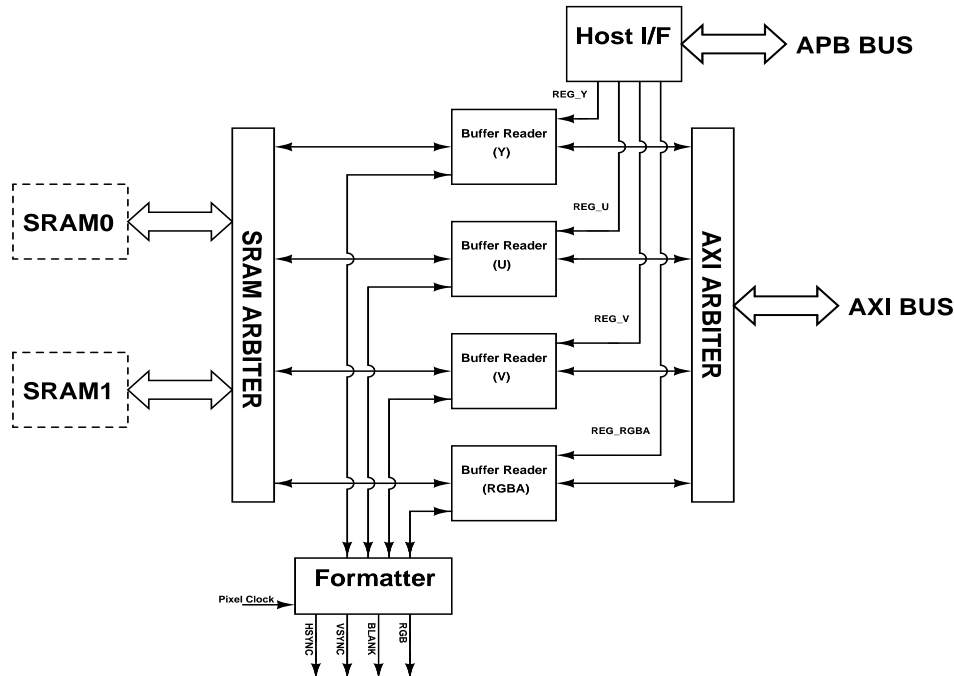

Figure 4.5: VDC block diagram

The *Host interface* provides an APB connection between the host CPU and the VDC internal register files. It supports a 32-bits data bus to access control and status registers. Host interface is able to generate an IRQ request in order to report an event to the CPU.

The *Buffer Reader* translates pixel virtual addresses to physical addresses using BPL. It accesses external memory via an arbiter connecting VDC to 64-bits AMBA AXI bus. Buffer Reader handles external memory (SDRAM) and internal memory (SRAM) accesses using 3 address generation modules. Worth mentioning, Buffer Readers operate independently from each other and contain their own address translation unit.

Display Formatter partially operates in a pixel clock domain and generates standard High-Definition Multimedia Interface (HDMI) input format. It's the DFMT's

responsibly to drive synchronization signals including vertical, horizontal and blanking signals (See Appendix regarding details about HDMI standard).

Rest of this chapter covers implementation details of Buffer Reader and Display Formatter followed by the VDC software approach to handle a sequence of frames.

Buffer Reader

The Buffer Reader provides an efficient way of translating virtual addresses without CPU intervention, accessing external memory and controlling pixel FIFO queues. Figure 4.6 illusterates block diagrams inside the Buffer Reader.

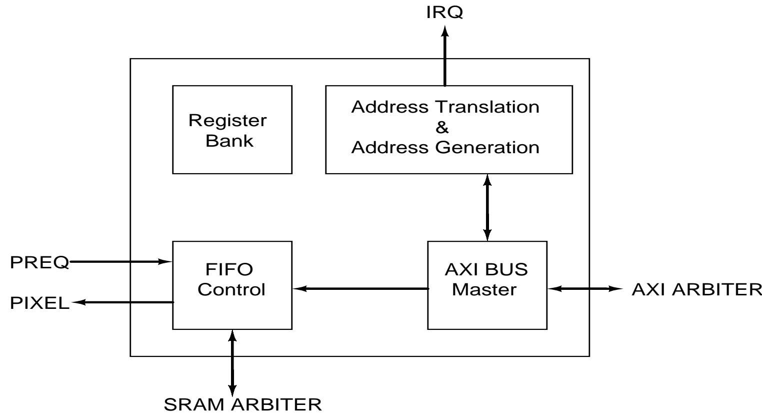

Figure 4.6: Buffer Reader block diagram

Register Bank holds vital information for translation and address calculation. It includes Buffer definitions (BPLA, BPLS, BS), OFFSET register and size of the window that is going to be display (XSIZE,YSIZE). The AXI Bus Master module maintains access to the external memory in privileged non-secure mode. In order to prioritize shared memory, the Buffer Reader doesn't employ master-induced wait states on the bus. The FIFO control module maintains read and write operation from/to SRAM by receiving combinational grant signal from the SRAM arbiter.

To enable the VDC to display variant window sizes starting at any 64-bit aligned addresses, first the host CPU loads the Buffer Reader with 32-bit OFFSET register. OFFSET defines the virtual address of the the first pixel that is going to be displayed inside the buffer using Formula 4.11:

$$OFFSET = (Offset\,Y \times\, Stride + Offset\,X) \times Unit\,Size + Original\,Page\,Offset \tag{4.11}$$

Where Offset X and Offset Y- are the upper left coordinates of the window inside the buffer that is going to be displayed on the screen. In case of showing an entire buffer, offset values in both directions are equal to zero.

By loading buffer parameters from the host interface, the Buffer Reader begins to generate 32-bit pixel virtual addresses starting at OFFSET. Each virtual address is logically divided into a page number and a page offset. For translating a virtual address, a physical base address of the page which requested pixel resides in is required. Using page number to index BPL and then issuing 1 beat burst length to access location BPLA + (8 $\times$ page number) retrieves the physical page base address of the desired pixel. This physical page base address concatenates with the page offset making pixel physical address.

The buffer read continues by issuing outstanding incremental bursts and stops either by the next page address translation or FIFO queue gets full. It again proceeds after a page translation or a dequeue request from the DFMT. Ultimately the Buffer Reader informs the controller unit once all the pixel addresses inside the window are issued and sent to the external memory.

The Buffer Reader issues maximum possible burst length (16 beats) to maximize the bus utilization except in 3 following conditions:

- End of a row : to support any window size
- End of a page : to avoid splitting over 4 KiB page boundary
- FIFO overflow : to prevent FIFO overflow if there is less than 16 beats space

The FIFO control unit is able to differentiate between pixel values and page base addresses. Since the VDC supports outstanding transactions and various window sizes, address generation unit saves number of issued bursts inside the page in order to signal FIFO control to ignore the next coming data from the external memory.

Size of the FIFO queue is parameterized by a peak latency in the system. The FIFO control performs write operation in 2 and read operation in 3 clock cycles. Additional registers placed on SRAM ports to avoid any timing problem. While write operation is always granted, read operation can stall for one clock cycle.

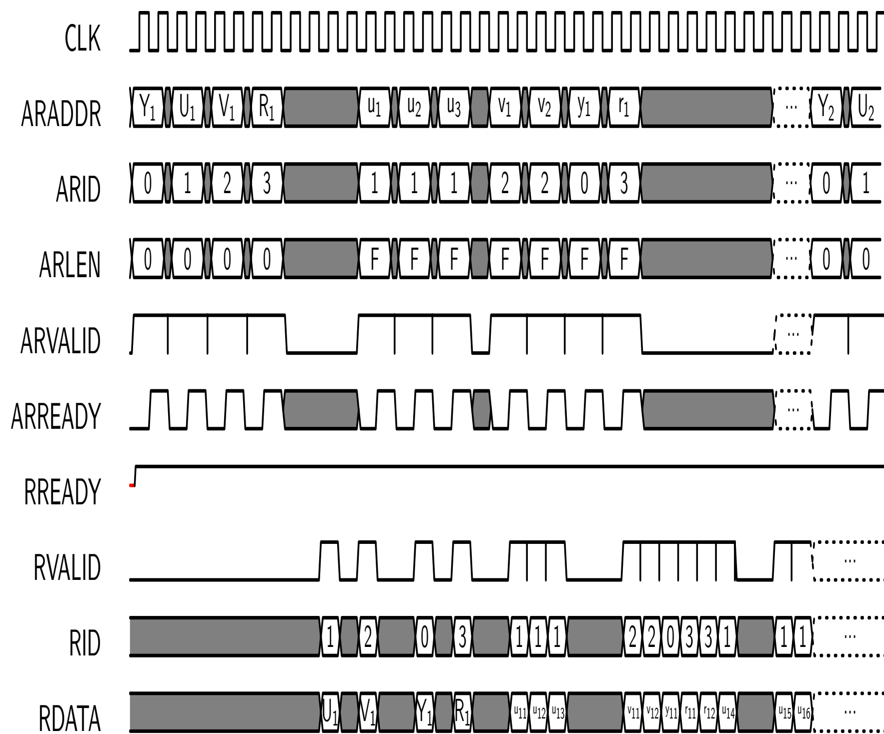

Figure 4.7: VDC AXI timing diagram

Figure 4.7 shows timing diagram of the VDC by connecting 4 Buffer Readers via an arbiter to the AXI bus.

Capital letters define bursts accessing BPL and small ones define bursts inside a page. Observing timing diagram presents decent features of AXI including independency of address and data channel, outstanding and out of out order burst transcations and mixture of Y, U, V and RGBα values on the RDATA. Worth mentioning, issuing bursts inside a page do not start until first physical address of a page retrieved from the BPL.

Display Formatter

The DFMT maintains connection of the VDC to the world outside namely a VGA/LCD display. It feeds the HDMI transmitter by generating VSYNC, HSYNC and blanking signals running on the pixel clock frequency. The DFMT uses 16-bit registers including display position (X0,Y0) on the screen and window size (XSIZE,YSZIE) to request pixels from the Buffer Readers. (Figure 4.8)

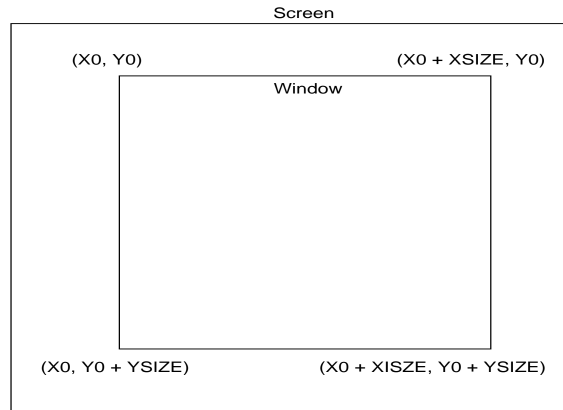

Figure 4.8: Borders of a displaying window on a screen

By adopting UMA, the Buffer Readers are not allowed to employ master-induced wait states on the system bus. Therefore, write requests to the FIFO queues must be always granted. In the other hand, VDC's latency-critical characteristic compels the DFMT's read requests to be also granted instantly. Thus, double single port SRAMs are used to image a memory that allows simultaneous read and write accesses on the FIFO queues. However, SRAM arbiter always prioritizes write requests and stalls read ones in the case of simultaneous access to the same SRAM.

To prevent dropping any single pixel on screen, first only one additional register is placed after SRAM RDATA register to diminish effect of stalling read requests. Depending on the pixel clock frequency and color depth, this one register was expected to sustain for particular number of clock cycles. But after observance of AXI arbiter behavior using ID bits for arbitration, fading of decent zigzag property for read and write on LSB mapped SRAMs leaked out. To overcome the problem, one extra register is added for each Buffer Reader resulting 12 registers in total counting those SRAM RDATA registers.

In order to avoid metastability and proper request scheme from the DFMT to the Buffer Reader, an interface logic is placed to connect these two clock domains.

Figure 4.9 shows an interface circuit between the DFMT and every Buffer Reader. Due to absence of clock synchronization module inside the VDC to register pixel values coming from Buffer Reader, current design only supports multiple 2 of pixel frequency.

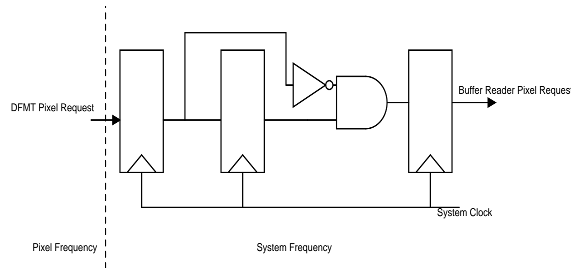

Figure 4.9: Interface circuit between DFMT and Buffer Reader pixel request

The DFMT waits until the FIFO queues contain sufficient number of pixels to display in order to avoid underflowing condition in a case of having initial high peak latency in the first frame. Additionally, the DFMT is able to generate a test frame when there is no activity on the AXI bus.

User Interface and Software approach

A C-based user interface maintains initialization, access and running the VDC from the host CPU. Using 2 kByte register file inside the VDC enables CPU to create two concurrent jobs to handle double buffering. A Linux device driver is developed in order to communicate with the VDC from CPU side. Software architecture blocks are shown in Figure 4.10.

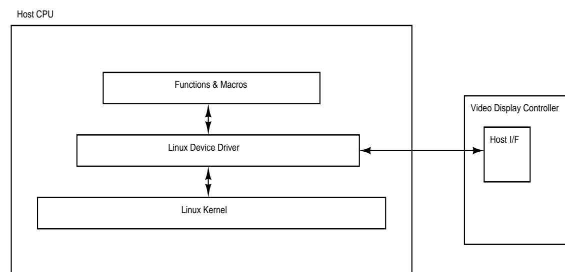

Figure 4.10: Software architecture blocks

Employing a modular design with the generic functions in both software and hardware sides facilities addition of extra layers, support of different display resolutions and scaling the design for cost and performance.

4.3 Verification

The VDC development process demands concurrent implementation of both hardware and software. Therefore, incorporating co-design and co-verification strategies from the early system design phase was inevitable. Having this in mind, precise register selections were made in order to enable user-friendly connection of the software and hardware parts. Due to importance of the bus behavior and its latency on the VDC design, a simulated behavior of average and peak bus latency were used to interface the VDC to the system shared memory. In order to feed the appropriate file formats to the design and verifying the results Matlab scripts are developed. The test sequence of Tractor for RGBα layer and Riverbed for YUV layer captured at 30 frame per seconds [13] with two resolutions of 640×480 pixels and 1920×1080 were used.
First, a paging scheme pages a frame with its two layers to 4 KiB pages and creates a page table for every Y,U,V and RGBα buffer. Page tables are written in a specific dynamic location in the shared system memory. Then, software program through device driver loads the VDC registers for each buffer with proper values and gives the control of the hardware to the VDC itself. In order to verify the design, VDC's output is written in a binary file and compares with the initial sequence of frames using Matlab.

Using in-house developed environment to simulate CPU and AXI/APB bus behavior enables to commence verification process from the first cycle of the design. Moreover, developing an AXI timing adapter module permits to test the peak latency constraint on the VDC.

VDC functionality was verified by applying 333 MHz and 166 MHz system and pixel (without blanking) clock frequencies targeting 300 MHz and 150 MHz clock frequencies with 11% performance margin, placing 28 KiB SRAM (16 KiB for RGBα and 12 KiB for YUV) and feeding two layers with VGA resolution. Assigning 8 outstanding burst credit for the VDC and sweeping mean time value between assertion of AVALID/AREADY and the first RVALID resulted 248 clock cycle average bus latency. Slight difference between nominal expected average latency (292 clock cycle) and the obtained value is originated from having smaller burst length in few cases like end of a row or end of a page which the VDC doesn't issue 16 beats full burst.

Peak latency constraint was verified by connecting the AXI interface of the VDC to the timing adapter. AXI timing adapter simply generates ON and OFF pulse signals and only grants the VDC to access the shared memory during ON cycles. Following above mentioned conditions, expected peak latency of 8192 clock cycles is fully accomplished.

4.4 Synthesis

As mentioned in section 4.1, the VDC performance does not improve as far as its required minimum bandwidth met. But due to integration policies, the VDC has been targeted for a system clock frequency.

Synthesis has been done for IBM's 65 nm low power CMOS process using Synopsys Design Compiler and constraining design with 300 MHz with performance margin of 11% and 1.5 ns input/output delay.

Area

Table 4.1 shows estimated areas for different blocks in μm^2 and their distributions inside the VDC. Synthesis tool reports a total area of 109318 μm^2. As an estimation of a gate count using average size of 1.4 μm^2 for each gate, number of gate counts results in 78 kGates.

Table 4.1: Modules area distribution

Module Name	**Area(μm^2)**	**Percentage(%)**
RGBα Buffer Reader	16535	15.1
Y Buffer Reader	16466	15.1
U Buffer Reader	16434	15.0
V Buffer Reader	16567	15.1
Controller	7865	7.2
Formatter	7394	6.7
AXI Arbiter	827	0.7
SRAM Arbiter	5633	5.1
Host Interface	20994	19.2

As table 4.1 implies host module which contains buffers register bank takes a major part of the area in comparison with the others components. The total area of the VDC including 28 KiB SRAM for the FIFO buffers (0.2 mm^2) is 0.31 mm^2.

Critical path

Critical path lies in AXI address generator module that makes the 32-bit pixel virtual address. Having been faced with the timing violation in the first try, decent feature of AXI outstanding burst mode aids to postpone the address generation for one extra clock cycle without any impact on the system performance. Therefore, timing met without using any multiple threshold voltage cells for synthesis.

Critical path does not act as the bottleneck for the entire design. Mainly shared

system memory latency limits the VDC performance by tardily pixel feeding that makes the image breaks on the screen.

Power estimation

Synthesis tool reports the leakage power of 3.4 μW in a room temperate (25 °C). Since the VDC is targeted for the embedded applications, careful decisions regarding power consumption were made to keep the power dissipation as low as possible. Main feature of the current VDC design that makes it viable for low-power applications is usage of single ported SRAM instead of common approach in the most VDC designs to use dual-port SRAM for implementing the pixel FIFO.

Conclusion

The proposed Video Display Controller is a synthesizable core intended to drive a VGA/LCD screen in an embedded environment. The controller is capable of connecting to the shared system memory using an AMBA AXI interface with support for the virtual memory accesses. In order to hide the high peak latency of the system bus, double dedicated single port SRAMs are used as a pixel FIFO buffer. But whatever size of the FIFO buffer is, a long-term average latency requirement compels a support for outstanding and out-of-order transactions.

Picture-in-picture feature accepting two input formats namely YUV444 and RGBα with a programmable display resolution has been implemented. To enhance memory management, configurable stride and non-page aligned access for all frame buffers is considered and implemented. To support variant resolutions, timing parameters including horizantal/vertical front porch, back porch and SYNC intervals are all customizable.

An AMBA APB interface connects the video display controller to the host CPU for programming registers, reporting events and controlling the entire display process. To avoid artifacts such as flicker and tearing, double buffering technique is applied. A C-based user interface and debug environment with a Linux device driver is developed to realize all mentioned features.

Appendix A

Appendix

A.1 Y'CbCr color space

Video displayers are driven by red, green and blue voltage signals (RGB) whereas storage and transmission of image inside chips are performed using Y'CbCr format. Y'CbCr is a color space composed of one luminance (luma) channel and two chrominance (chroma) channels. Luminance carries brightness and chrominance carries color information. Considering the fact that human eye is more sensitive to luminance changes rather than chrominance, *chroma subsampling* is often exploited in order to reduce resolution for chroma channel. This dramatically reduces the required bandwidth for each frame.

Variant subsampling ratio may be utilized to reduce the chroma resolution. YUV420 is a common subsampling ratio in which both horizontal and vertical directions are halved. Hence, every four pixels on screen shares same Cb and Cr values. Beside, YUV420 color components may store in 3 separates plane resulting YUV420 with 3-planes or luminance channel stores in one plane and chrominance channels interleaved and stored in another plane resulting YUV420 with 2-planes.

The conversion between Y'CbCr and RGB color space are realized by 3*3 matrix multiplications. Equation A.1 and A.2 show Y'CbCr conversion to RGB and reverse operation respectively[14].

$$\begin{pmatrix} Y' \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.1687 & -0.3313 & 0.5 \\ 0.5 & -0.4187 & -0.0813 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix} \quad (A.1)$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1.402 \\ 1 & -0.34414 & -0.71414 \\ 1 & 1.772 & 0 \end{pmatrix} \begin{pmatrix} Y' \\ Cb - 128 \\ Cr - 128 \end{pmatrix} \tag{A.2}$$

A.2 HDMI standard

High-Definition Multimedia Interface (HDMI) is a standard interface for connecting display controller to VGA/LCD display. HDMI provides all-digital audio/video interface on a single cable replacing consumer analog standard, VGA. HDMI is able to transmit uncompressed high-definition video plus digital audio. Today HDMI video interface is incorporated on everything from set-top boxes and DVD players to phones and even cameras. HDMI uses Transition Minimized Differential Signaling (TMDS) for transmitting video,audio and auxiliary data. TMDS link includes three TMDS Data channels and a single TMDS Clock channel[15].

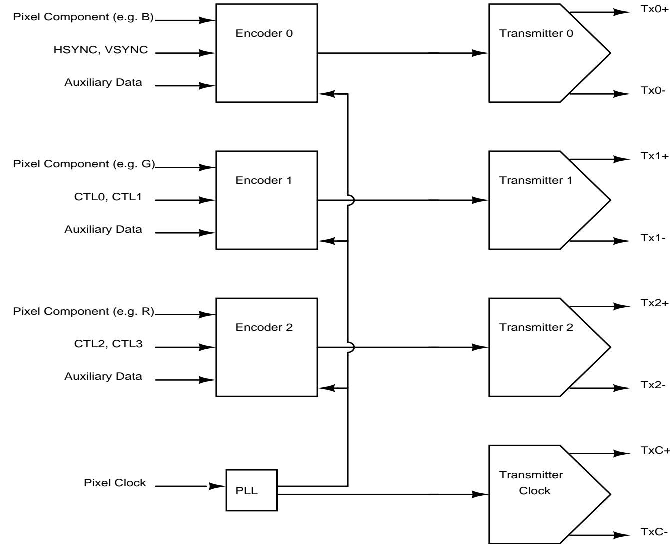

Figure A.1: HDMI

Since HDMI link operates in three modes (Video Data Period, Data Island period, and Control period), the values of CTL0, CTL1, CLT2 and CLT3 indicate the type of upcoming data period[15].

Bibliography

[1] Michael J. Flynn and Wayne Luk. *Computer System Design: System-on-chip.* May 2011. unpublished.

[2] Sudeep Pasricha and Nikil Dutt. *On-Chip Communication Architectures: System on Chip Interconnect.* Morgan Kaufmann, 2008.

[3] ARM Limited. *AMBA Specification (Rev.2)*, 1999.

[4] ARM Limited. *AMBA AXI Protocol specification (Rev.1)*, 2004.

[5] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantative Approach.* Elsevier, fourth edition, 2006.

[6] Avi Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts.* John Wiley and Sons Inc, seventh edition, 2004.

[7] ARM Limited. *ARM Architecture: ARMv7-A and ARMv7-R edition*, 2009.

[8] Mel Gorman. *Understanding the Linux Virtual Memory Manager.* Prentice Hall, 2004.

[9] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly Media, Sebastopol, CA, third edition, 2006.

[10] William Stallings. *Operating Systems: Internals and Design Principals.* Prentice Hall, fifth edition, 2005.

[11] Open source material. *ARM Linux 2.6.11 source code.* 2002. `http://lxr.linux.no/#linux+v2.6.11/include/asm-arm/pgtable.h`.

[12] Ashley Stevens. *QoS for High-Performance and Power-Efficient HD Multimedia.* ARM Limited, 2010.

[13] Dr. Karl Mauthe (Taurus Media Technik). Tractor, riverbed. 2001. Video seuqences recorded using Sony HDW-F900 camera.

[14] Per Edgren. Vlsi implementation of a video encoder noise-reduction pre-filter. Master's thesis, Lund University, October 2010.

[15] LLC HDMI Licensing. *High-Definition Multimedia Interface (Specification Version 1.3)*, 2006.