Memory Hierarchy for Microblaze and PowerPC based Systems

A dissertation submitted in partial fulfillment of the requirements

for the degree of

Master of Technology

In

Computer Application

By

Nikunj Shroff

(2005JCA2435)

Under the guidance of

Dr. Kolin Paul

(Department of Computer Science and Engineering)

Indian Institute of Technology Delhi

May 2007

CERTIFICATE

This is to certify that the project entitled “Memory Hierarchy for Microblaze and PowerPC based Systems” submitted by Nikunj Shroff in partial fulfillment of the requirement for the award of the degree of Master of Technology in Computer Applications to the Indian Institute of Technology, Delhi, is a record of bona-fide work carried by him under my supervision and guidance.

Dr. Kolin Paul

Department of Computer Science and Engineering

Indian Institute of Technology, New Delhi

ACKNOWLEDGEMENT

It is my privilege and great pleasure to convey my gratitude to those who have directly or indirectly, helped me in this work. I would like to express my gratitude to Dr. Kolin Paul for his keen observation, help and guidance regarding my work throughout the semester. I am thankful to him for the constant encouragement and advice he gave me at different steps of our literature survey, problem understanding and its implementation. I am also thankful to Prof. M. Balakrishnan for showing keen interest in solving critical problems in this project.

Nikunj Shroff
(2005JCA2435)

Contents

Abstract

Embedded systems and general-purpose computers differ in many ways. One key difference is how programs are loaded and executed out of memory. Software written on personal computers is typically compiled into a completely re-locatable format with no absolute addresses embedded in the instructions. To access the program, the operating system finds available memory; the loader (an operating system function) retrieves the program, inserts all of the necessary absolute addresses for the program, and the program are put into available memory.

Ideally one would desire an indefinitely large memory capacity such that any particular word would be immediately available. We are trying to analyze the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding and which has higher access time. We are trying to explore the possibility of characterizing a memory hierarchy for embedded system, which uses Microblaze and PowerPC processor in order to make efficient use of our application running on these systems.

1. Introduction

We are presently working on the FPGA board which is manufactured by Xilinx and the specifications of the board is XC2VP30, Grade ff896, Speed -7.

A field programmable gate array (FPGA) [3] is a semiconductor device containing pogrammable logic components and programmable interconnects. The programmable logic components can be programmed to duplicate the functionality of basic logic gates such as AND, OR, XOR, NOT or more complex combitional functions such as decoders or simple math functions. In most FPGAs, these programmable logic components (or logic blocks, in FPGA parlance) also include memory elements, which may be simple flip flops or more complete blocks of memories.

A hierarchy of programmable interconnects allows the logic blocks of an FPGA to be interconnected as needed by the system designer, somewhat like a one-chip programmable. These logic blocks and interconnects can be programmed after the manufacturing process by the customer/designer (hence the term "field programmable") so that the FPGA can perform whatever logical function is needed.

The **MicroBlaze** embedded soft core is a reduced instruction set computer (RISC) optimized for implementation in Xilinx field programmable gate arrays (FPGAs).

Features :

The MicroBlaze embedded soft core is highly configurable, allowing users to select a specific set of features required by their design. The processor’s fixed feature set includes the following:

. Thirty-two 32-bit general purpose registers

. 32-bit instruction word with three operands and two addressing modes

. 32-bit address bus

. Single issue pipeline

The **PowerPC** [7] architecture is a RISC architecture, and as such has been designed to facilitate the design of processors that use pipelining and parallel execution units to maximize instruction throughput. However, the PowerPC architecture does not define the internal hardware details of an implementation.

The PowerPC is a 32-bit implementation of embedded-environment architecture. The PowerPC architecture provides a software model that ensures compatibility between implementations of the PowerPC family of microprocessors. The PowerPC architecture defines parameters that guarantee compatible processor implementations at the application-program level, allowing broad flexibility in the development of derivative PowerPC implementations that meet specific market requirements.

1.1 MicroBlaze Processor Programs and Memory

In MicroBlaze, we can write either C, C++, or Assembly programs, and use the Embedded Development Kit (EDK) to transform our source code into bit patterns stored in the physical memory of a EDK System. Our programs typically access local and on-chip memory, external memory, and memory-mapped peripherals. Memory requirements for our programs are specified in terms of how much memory is required for storing the instructions and how much memory is required for storing the data associated with the program.

MicroBlaze address space is divided between the system address space and user address space. In certain examples, we need advanced address space management, which we can do with the help of linker a script.

1.2 Current Address Space Restrictions

1.2.1.Memory and Peripherals Overview

MicroBlaze uses 32-bit addresses, and as a result it can address memory in the range 0 through 0xFFFFFFFF. MicroBlaze can access memory either through its Local Memory Bus (LMB) port or through the On-chip Peripheral Bus (OPB). The LMB is designed to be a fast access, on-chip block RAM (BRAM) memories only bus. The OPB represents a general purpose bus interface to on-chip or off-chip memories as well as other non-memory peripherals.

1.2.2. BRAM Size Limits

The amount of BRAM memory that can be assigned to the LMB address space or to each instance of an OPB mapped BRAM peripheral is limited. The largest supported BRAM memory size for Virtex-II it is 64 kB. It is important to understand that these limits apply to each separately decoded on-chip memory region only. The total amount of on-chip memory available to a MicroBlaze system might exceed these limits. The total amount of memory available in the form of BRAMs is also FPGA device specific. Smaller devices of a given device family provide less BRAM than larger devices in the same device family.

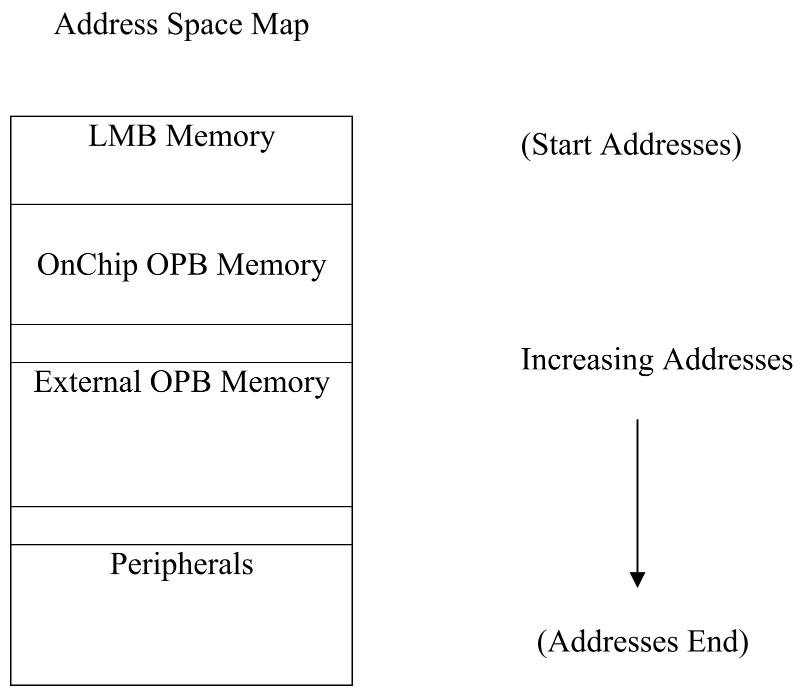


Figure 1.1 Address Mapping in Microblaze system.

1.2.3. Special Addresses

Every MicroBlaze system must have user-writable memory present in addresses 0x00000000 through 0x00000028. These memory locations contain the addresses that MicroBlaze jumps to after a reset, interrupt, or exception event occurs. This memory can be part of the LMB or the OPB BRAM address space.

1.2.4. OPB Address Range Details

Within the OPB address space, you can arbitrarily assign address space to on/off-chip memory peripherals and to on/off-chip non-memory peripherals. The OPB address space might contain holes representing regions that are not associated with any OPB peripheral. Special linker scripts and directives might be required to control the assignment of object file sections to address space regions.

1.2.5. Address Map

The MicroBlaze Hardware Specification (MHS) file contains an address map specifying the addresses of LMB memory, OPB memory, external memory, and peripherals. The address range grows from 0. At the lowest range is the LMB memory. This is followed by the OPB memory, external memory and the Peripherals. Some addresses in this address space have predefined meaning. The processor jumps to address 0x0 on reset, to address 0x8 on exception, and to address 0x10 on interrupt.

1.3 Memory Speeds and Latencies

MicroBlaze requires **two clock cycles to access on-chip Block RAM** connected to the LMB for write and **two clock cycles for read**. **On chip memory connected to the OPB bus requires three cycles for write and four cycles for read**. External memory access is further limited by off-chip memory access delays for read access, **resulting in five to seven clock cycles for read**. Furthermore, memory accesses over the OPB bus might incur **further latencies due to bus arbitration overheads**. As a result, instructions or data that must be accessed quickly should be stored in LMB memory when possible.

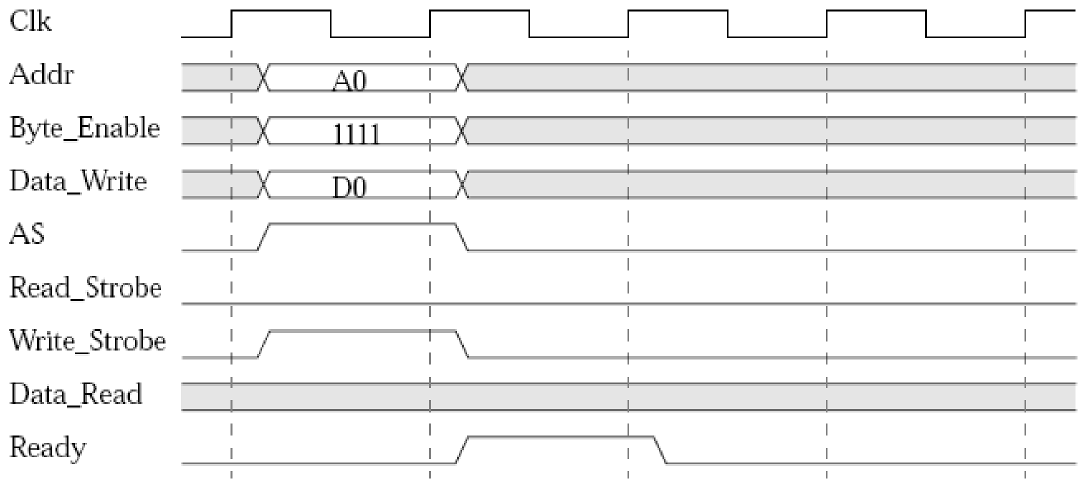


Figure 1.2 LMB Generic Write Operation [2]

1.4 Object-File Sections

The sections of an executable file are created by concatenating the corresponding sections in an object (.o) file. Sectional Layout of an object or an Executable File [9].

The Object file is divided in following sections :

1. .text

This section of the object file contains executable code. This section has the `x` (executable), `r` (read-only) and `i` (initialized) flags.

2. .rodata

This section contains read-only data of a size more than a specified size; the default is 8 bytes. We can change the size of the data put into this section with the **mb-gcc -G** option. All data in this section is accessed using absolute addresses. This section has the r (read-only) and the i (initialized) flags.

3. .sdata2

This section contains small read-only data of size less than 8 bytes. We can change the size of the data going into this section with the **mb-gcc -G** option. All data in this section is accessed with reference to the read-only small data anchor. This ensures that all data in the .sdata2 section can be accessed using a single instruction; therefore, a preceding imm instruction is never necessary. This section has the r (read-only) and the i (initialized) flags.

4. .data

This section contains read-write data of a size more than a specified size; the default is 8 bytes. You can change the size of the data going into this section with the **mb-gcc –G** option. All data in this section is accessed using absolute addresses. This section has the w (read-write) and the i (initialized) flags.

5. .sdata

This section contains small read-write data of a size less than a specified size; the default is 8 bytes. You can change the size of the data going into this section with the **mb-gcc –G** option. All data in this section is accessed with reference to the read-write small data anchor. This ensures that all data in the .sdata section uses a single instruction; therefore, a preceding imm instruction is never necessary. This section has the w (read-write) and the i (initialized) flags.

6. .sbss

This section contains small un-initialized data of a size less than a specified size; the default is 8 bytes. You can change the size of the data going into this section with the **mb-gcc –G** option. This section has the w (read-write) flag.

7. .bss

This section contains un-initialized data of a size more than a specified size; the default is 8 bytes. You can change the size of the data going into this section with the **mb-gcc –G** option. All data in this section is accessed using absolute addresses. The stack and the heap are also allocated to this section. This section has the w (read-write) flag. The linker script describes the mapping between all of the sections in all of the input object files, and the output executable file.

Example :

BSS: Uninitialized Data (Read/Write)	DATA: Initialized Data (Read/Write)
RODATA: Constants (Read Only)	TEXT: Program Code (Read Only)

A simple C program shows how different entities can be segmented in the compilation process.

```
#include <stdio.h>
int array_data[10] = {0,1,2,3,4,5,6,7,8,9};            // DATA
const int array_const[10] = {9,8,7,6,5,4,3,2,1,0};     // RODATA
int main(){
int i;                                                 //BSS
i = i + 10;                                            //TEXT
printf("%d\n", array_data[0]);                         //TEXT
array_data[0] = array_const[0];                        //TEXT
printf("%d\n",array_data[0]);                          //TEXT
return 0;
}
```

2. Methodology used and work done

2.1 Microblaze

Data Cache and Instruction cache is integrated inside Microblaze through transparent links as shown in figure. This links are used to connect Microblaze to the ONChip available BRAM.

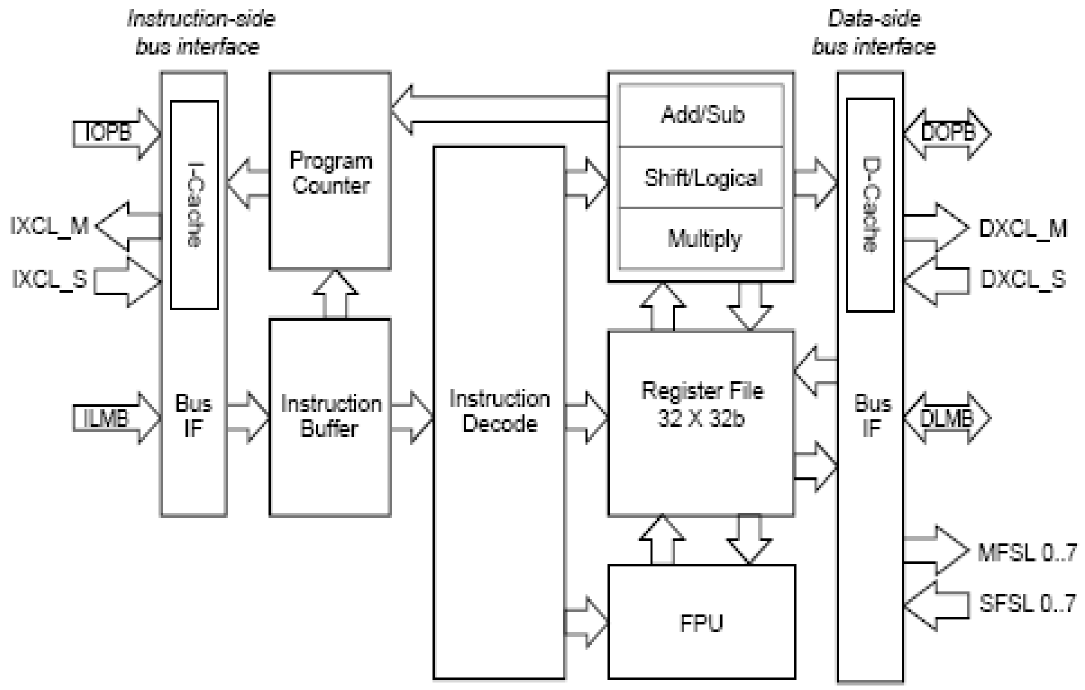


Figure 2.1 Block diagram of Core Architecture of Microblaze [2]

2.1.1 Instructions in Microblaze

All MicroBlaze instructions are 32 bits and are defined as either Type A or Type B. Type A instructions have up to two source register operands and one destination register operand. Type B instructions have one source register and a 16-bit immediate operand (which can be extended to 32 bits by preceding the Type B instruction with an IMM instruction). Type B instructions have a single destination register operand. Instructions are provided in the following functional categories: arithmetic, logical, branch, load/store, and special. List of MicroBlaze instruction set is as follows.

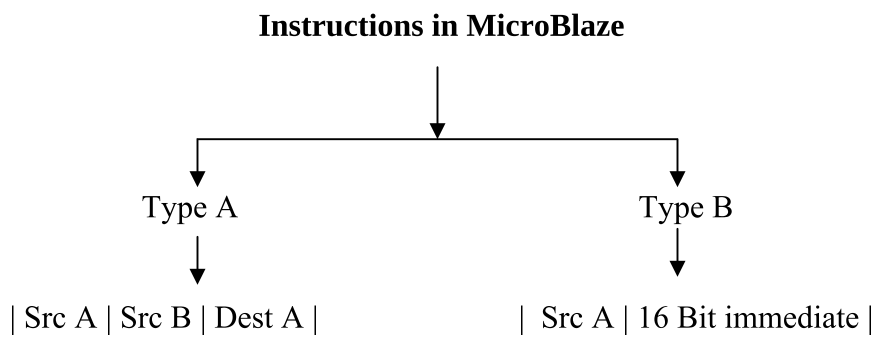

Instructions in MicroBlaze
Type A
Type B
| Src A | Src B | Dest A |
| Src A | 16 Bit immediate |

Figure 2.2 Basic Instruction type in Microblaze

Pipeline Architecture

MicroBlaze instruction execution is pipelined. The pipeline is divided into five stages:

1. Fetch (IF),
2. Decode (OF),
3. Execute (EX),
4. Access Memory (MEM),
5. Writeback (WB).

For most instructions, each stage takes one clock cycle to complete. Consequently, it takes five clock cycles for a specific instruction to complete, and one instruction is completed on every cycle. A few instructions require multiple clock cycles in the execute stage to complete. This is achieved by stalling the pipeline. When executing from slower memory, instruction fetches may take multiple cycles. This additional latency will directly affect the efficiency of the pipeline. MicroBlaze implements an instruction prefetch buffer that reduces the impact of such multi-cycle instruction memory latency. While the pipeline is stalled by a multi-cycle instruction in the execution stage the prefetch buffer continues to load sequential instructions. Once the pipeline resumes execution the fetch stage can load new instructions directly from the prefetch buffer rather than having to wait for the instruction memory access to complete.

Branches

Normally the instructions in the fetch and decode stages (as well as prefetch buffer) are flushed when executing a taken branch. The fetch pipeline stage is then reloaded with a new instruction from the calculated branch address. A taken branch in MicroBlaze takes three clock cycles to execute, two of which are required for refilling the pipeline. To reduce this latency overhead, MicroBlaze supports branches with delay slots.

Delay Slots

When executing a taken branch with delay slot, only the fetch pipeline stage in MicroBlaze is flushed. The instruction in the decode stage (branch delay slot) is allowed to complete. This technique effectively reduces the branch penalty from two clock cycles to one. Branch instructions with delay slots have a D appended to the instruction mnemonic. For example, the BNE instruction will not execute the subsequent instruction (does not have a delay slot), whereas BNED will execute the next instruction before control is transferred to the branch location.

A delay slot must not contain the following instructions: IMM, branch, or break. Interrupts and external hardware breaks are deferred until after the delay slot branch has been completed.

Instructions that could cause recoverable exceptions (e.g. unaligned word or half word load and store) are allowed in the delay slot. If an exception is caused in a delay slot the ESR[DS] bit will be set, and the exception handler is responsible for returning the execution to the branch target (stored in the special purpose register BTR) rather than the sequential return address stored in R17.

2.1.2 Memory Architecture

MicroBlaze is implemented with a Harvard memory architecture, i.e. instruction and data accesses are done in separate address spaces. Each address space has a 32 bit range (i.e. handles up to 4 GByte of instructions and data memory respectively). The instruction and data memory ranges can be made to overlap by mapping them both to the same physical memory. The latter is useful e.g. for software debugging.

Both instruction and data interfaces of MicroBlaze are 32 bit wide and use big endian, bitreversed format. MicroBlaze supports word, halfword, and byte accesses to data memory. Data accesses must be aligned (i.e. word accesses must be on word boundaries, halfword on halfword bounders), unless the processor is configured to support unaligned exceptions. All instruction accesses must be word aligned.

MicroBlaze does not separate between data accesses to I/O and memory (i.e. it uses memory mapped I/O). The processor has up to three interfaces for memory accesses: Local Memory Bus (LMB), On-Chip Peripheral Bus (OPB), and **Xilinx CacheLink (XCL).** The LMB memory address range must not overlap with OPB or XCL ranges. **MicroBlaze has a single cycle latency for accesses to local memory (LMB) and for cache read hits**. **A data cache write normally has two cycles of latency (more if the posted-write buffer in the memory controller is full).**

2.1.3 Instruction Cache

MicroBlaze may be used with an optional instruction cache for improved performance when executing code that resides outside the LMB address range. The instruction cache has the following features:

Direct mapped (1-way associative)

User selectable cacheable memory address range

Configurable cache and tag size

Caching over CacheLink (XCL) interface

Option to use 4 or 8 word cache-line

Cache on and off controlled using a bit in the MSR

Optional WIC instruction to invalidate instruction cache lines

General Instruction Cache Functionality

When the instruction cache is used, the memory addresses space in split into two segments: a cacheable segment and a non-cacheable segment.
The cacheable segment is determined by two parameters: C_ICACHE_BASEADDR and C_ICACHE_HIGHADDR. All addresses within this range correspond to the cacheable address segment. All other addresses are non-cacheable.

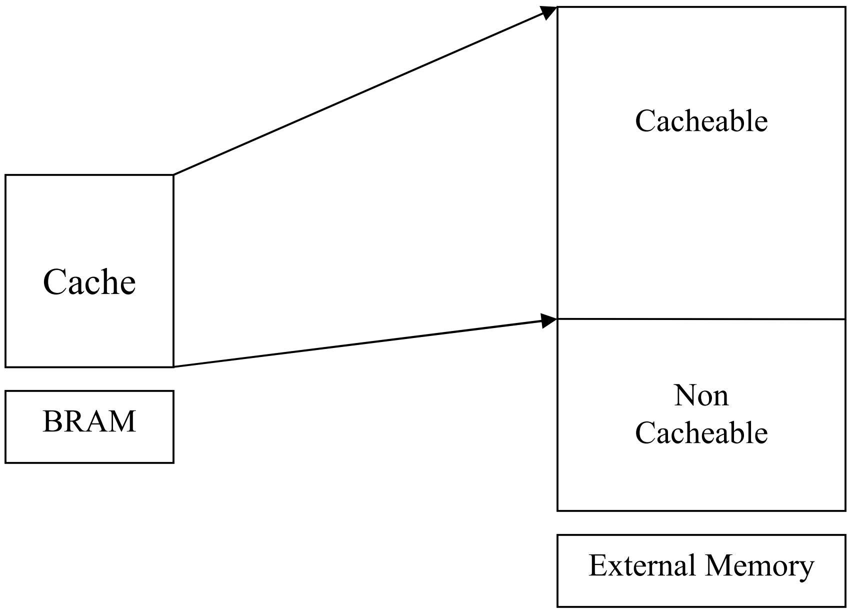


Figure 2.3 Cache Functionality

The cacheable instruction address consists of two parts: the cache address, and the tag address. **The MicroBlaze instruction cache can be configured from 2kB to 64 kB**. This corresponds to a cache address of between 11 and 16 bits. The tag address together with the cache address should match the full address of cacheable memory.

Example:

MicroBlaze configured with

C_ICACHE_BASEADDR= 0x00300000,

C_ICACHE_HIGHADDR=0x0030ffff,

C_CACHE_BYTE_SIZE=4096, and

C_ICACHE_LINELEN=8;

Cacheable memory of 64 kB uses 16 bits of byte address, and the 4 kB cache uses 12 bits of byte address, thus the required address tag width is: 16-12=4 bits. The total number of block RAM primitives required in this configuration is: 2 RAMB16 for storing the 1024 instruction words, and 1 RAMB16 for 128 cache line entries, each consisting of: 4 bits of tag, 8 word-valid bits, 1 line-valid bit. In total 3 RAMB16 primitives.

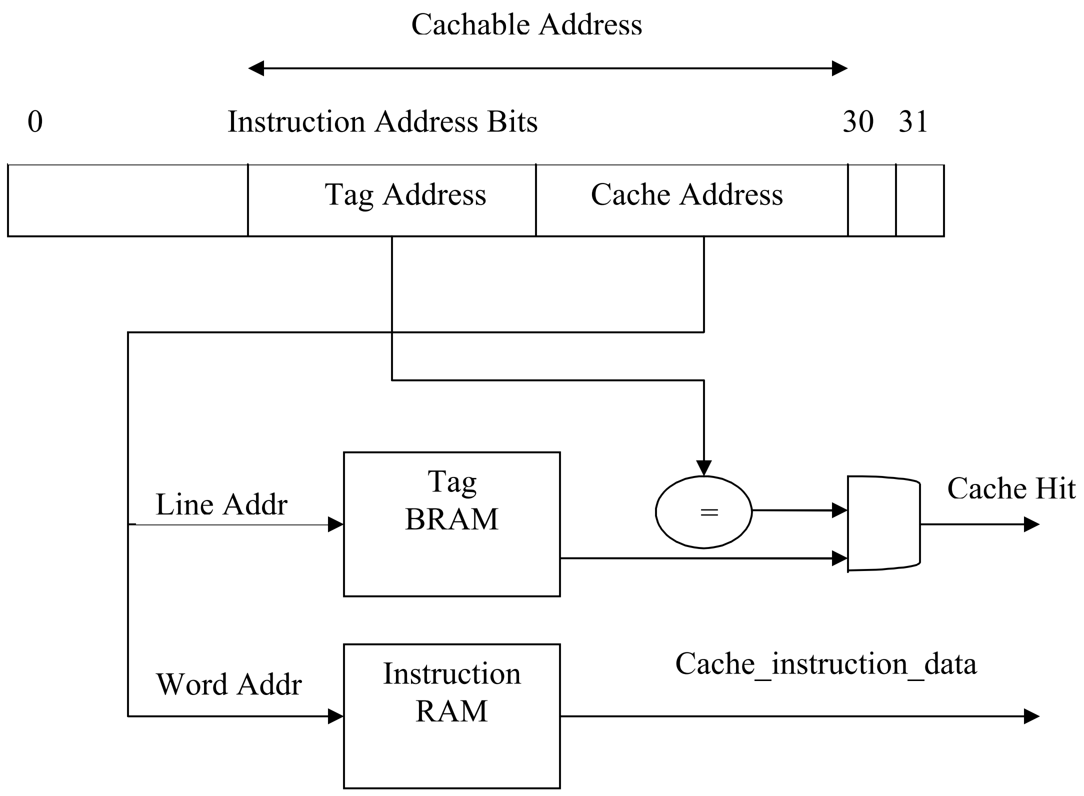


Figure 2.4 Instruction Cache organization

Instruction Cache Operation (For Microblaze v5.00a)

For every instruction fetched, the instruction cache detects if the instruction address belongs to the cacheable segment. If the address is non-cacheable, the cache controller ignores the instruction and lets the OPB or LMB complete the request. If the address is cacheable, a lookup is performed on the tag memory to check if the requested address is currently cached. The lookup is successful if: the word and line valid bits are set, and the tag address matches the instruction address tag segment. On a cache miss, the cache controller will request the new instruction over the instruction CacheLink (IXCL) interface, and wait for the memory controller to return the associated cache line.

Instruction Cache Operation (For Microblaze v4.00a or older)

For every instruction fetched, the instruction cache detects if the instruction address belongs to the cacheable segment. If the address is non-cacheable, the cache controller ignores the instruction and allows the OPB to fulfill the request. If the address is cacheable, a lookup is performed on the tag memory to check if the requested address is currently cached. The lookup is successful if: the valid bit is set, and the tag address matches the instruction address tag segment.

If the instruction is in the cache, the cache controller will drive the ready signal (Cache_Hit) and the cached instruction (Cache_instruction_data) to the pipeline. On a cache miss, the cache controller will wait until the missing instruction has been retrieved (either over the OPB, or the CacheLink interface depending on the caching scheme used), and then store it and its associated tag bits in the corresponding cache location.

The contents of the cache are preserved by default when the cache is disabled. The user may overwrite the contents of the cache using the WIC instruction or using the hardware debug logic of MicroBlaze.

Hardware Debug Logic

The HW debug logic can be used to perform a similar operation to the WIC instruction. Lock Bit. The lock bit can be used to permanently lock a code segment into the cache and therefore guarantee the instruction execution time. Locking of a cacheline can result in decreased cache performance, because the lock prevents the caching of all other instructions that map to the same cache location. In most cases adding instruction side LMB memory is a better choice for guaranteed access to certain code segments than cacheline locking. **The access latency of LMB BRAM is the same as for a cache hit.**

Instruction Cache Software Support

MSR Bit

The ICE bit in the MSR provides software control to enable and disable caches. The contents of the cache are preserved by default when the cache is disabled. The user can invalidate cache lines using the WIC instruction or using the hardware debug logic of MicroBlaze.

WIC Instruction (Write to instruction cache)

The optional WIC instruction (C_ALLOW_ICACHE_WR=1) is used to invalidate cache lines in the instruction cache from an application. WIC instruction should only be used when the instruction cache is disabled. **Latency is 1 cycle.** The WIC instruction may be used to update the instruction cache from a software program.

2.1.4 Data Cache

MicroBlaze may be used with an optional data cache for improved performance. The cached memory range must not include addresses in the LMB address range. The data cache has the following features

Direct mapped (1-way associative)

Write-through

User selectable cacheable memory address range

Cache on and off controlled using a bit in the MSR

For MicroBlaze V4.00 a

Configurable cache size and tag size

Caching over CacheLink (XCL) interface

Option to use 4 or 8 word cache-lines

Optional WDC instruction to invalidate data cache lines

For MicroBlaze V5.00 a

Configurable caching over OPB or CacheLink

4 word cache-line (only with CacheLink)

Individual cache line lock capability

Instructions to write to the data cache

Memory is organized into a cacheable and a non-cacheable segment

Data Cache Organization (Microblaze V 4.00a)

MicroBlaze can be configured to cache data over either the OPB interface, or the dedicated Xilinx CacheLink interface. The choice is determined by the setting of the two parameters:

C_USE_DCACHE and

C_DCACHE_USE_FSL

Caching over CacheLink uses 4 word cache lines (critical word first) for read misses.	OPB caches use single word cache lines.
CacheLink allows posted write accesses on write-misses.	OPB caches requires the write access to be completed before execution is resumed
CacheLink uses dedicated interface, instead of the OPB interface, for memory accesses. This reduces the traffic on the OPB. (MCH0 and MCH1)	No such dedicated interface available for OPB.
The CacheLink interface requires a specialized memory controller interface.	The OPB interface uses standard OPB memory controllers

Table 2.1 Comparision between OPB Cache and Xilinx cache link

2.1.5 Various models of memory for MicroBlaze

1. Cache not initialized

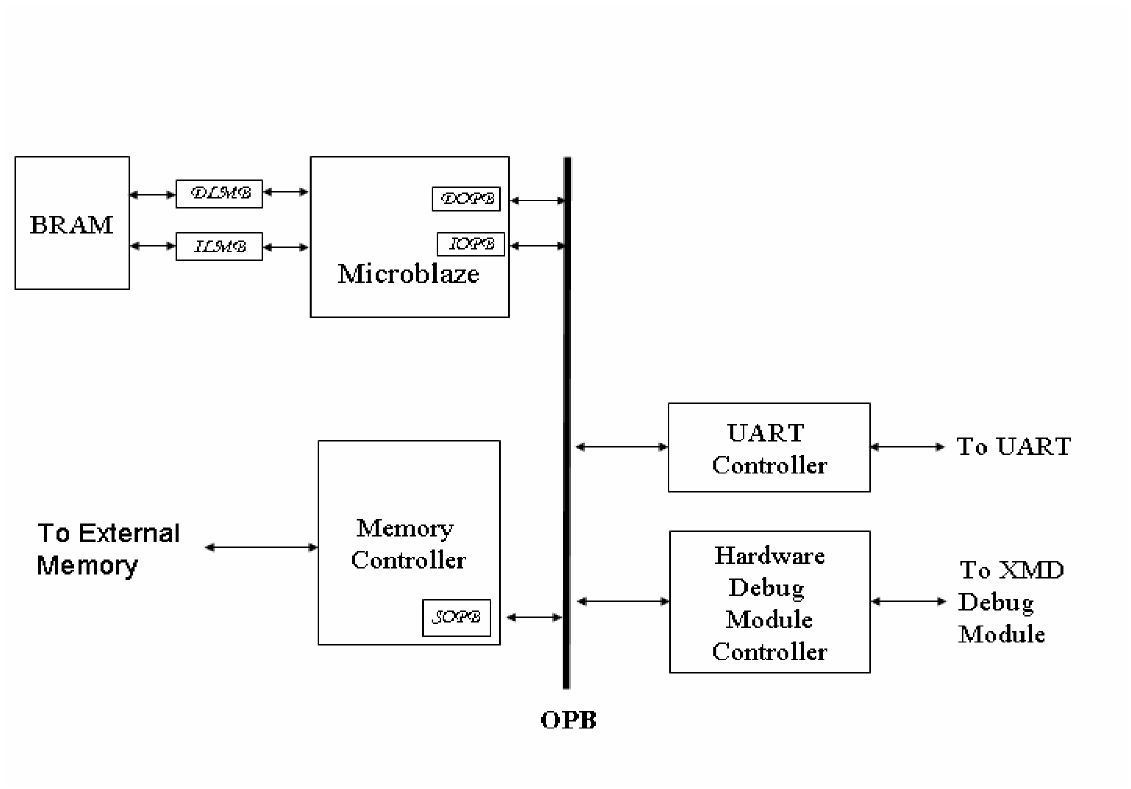


Figure 2.5 Microblaze design when cache not initialized.

This is a very basic model in which memory is connected through OPB using memory controller using slave mode. DOPB and IOPB is integrated in the SOPB link therefore data and instructions can be transferred from External memory to memory controller from where it moves to OPB Bus. Microblaze fetches instruction through IOPB and data through DOPB attached to OPB.

This model is simple but it requires more cycles for Microblaze to fetch data, as the instructions and data is transferred through OPB which is shared among all the peripherals connected to OPB.

2. Cache initialized on OPB bus.

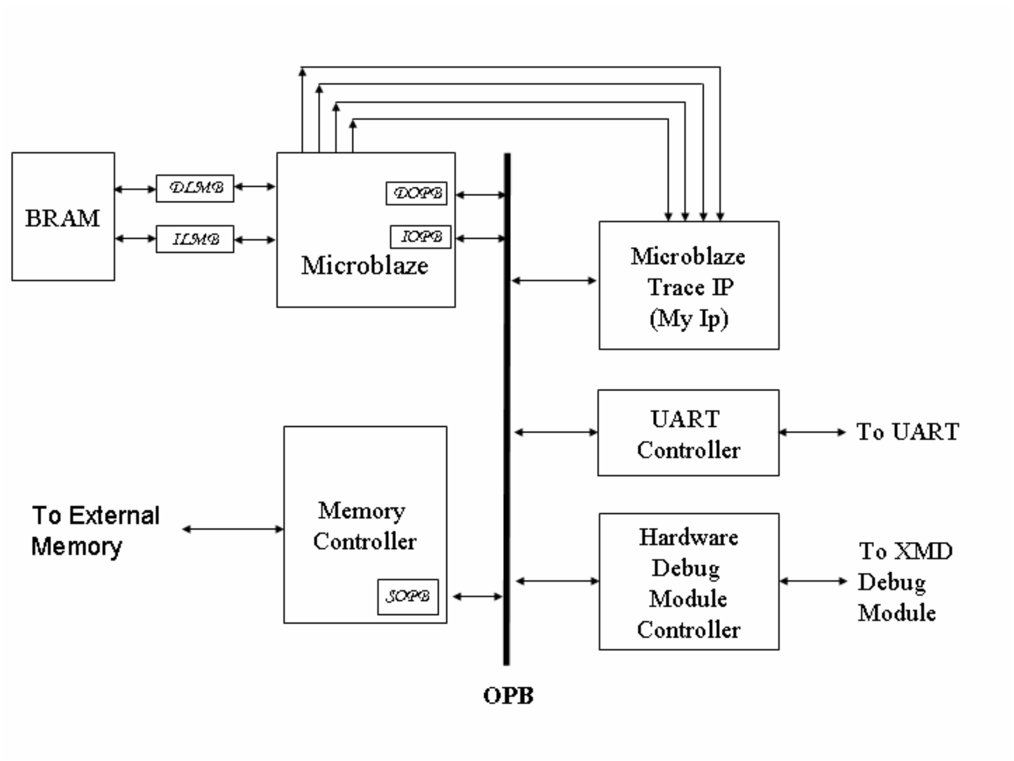


Figure 2.6 Microblaze design when caching is done through OPB Bus.

This is a one step ahead then the previous model in which memory is connected through OPB using memory controller using slave mode. DOPB and IOPB is integrated in the SOPB link therefore data and instructions can be transferred from External memory to memory controller from where it moves to OPB Bus. Microblaze fetches instruction through IOPB and data through DOPB attached to OPB. In this model cache is instantiated in the Microblaze which uses OnChip memory BRAM Blocks to store instruction and data.

This model requires less cycles as to fetch instructions and data as compared to previous one because once the instruction and data is fetched through OPB then it is copies in the cache and further if the instruction or data is needed then Microblaze copies it from the cache if it is available. If it is not available in the cache then it is transferred from OPB through IOPB and DOPB.

In this model instruction and data is transferred through OPB and/or cache

3. Cache initialized on Xilinx cache links.

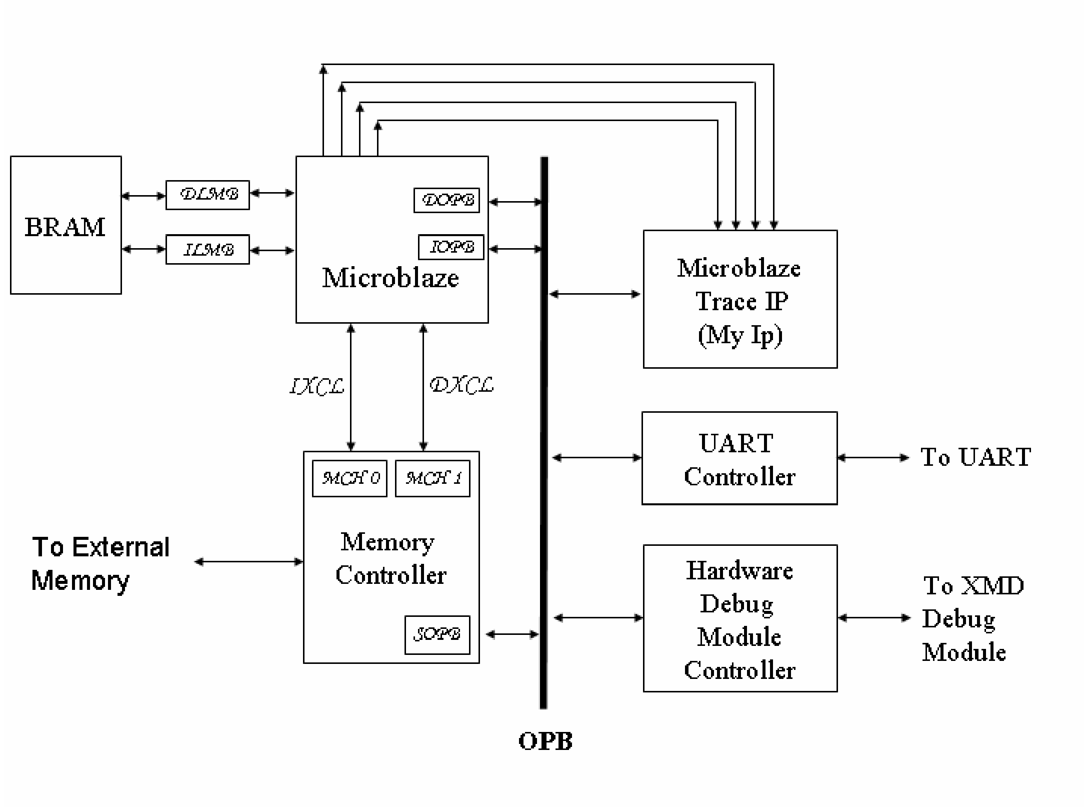


Figure 2.7 Microblaze design when caching is done using Xilinx Cache link

In the previous two models instruction and data was transferred through OPB. In previous models OPB bus was shared among all the peripherals including Microblaze and memory controller. So the overhead of transfer of instruction and data was quite more.

In this model a dedicated link is established between Microblaze and Memory controller through MCH0 and MCH1 [6] which is known as Xilinx cache link which is responsible of transferring data and instructions through these links. This model is better than previous one.

In this model instruction and data is transferred through ixcl and dxcl. On a cache miss or data miss, the cache controller will request the new instruction / data over the instruction CacheLink (IXCL) interface / Data CacheLink (DXCL) , and wait for the memory controller to return the associated cache line.

This model performs well because of less latency of access of instruction and data from memory controller.

4. Cache initialized on FSL cache links.

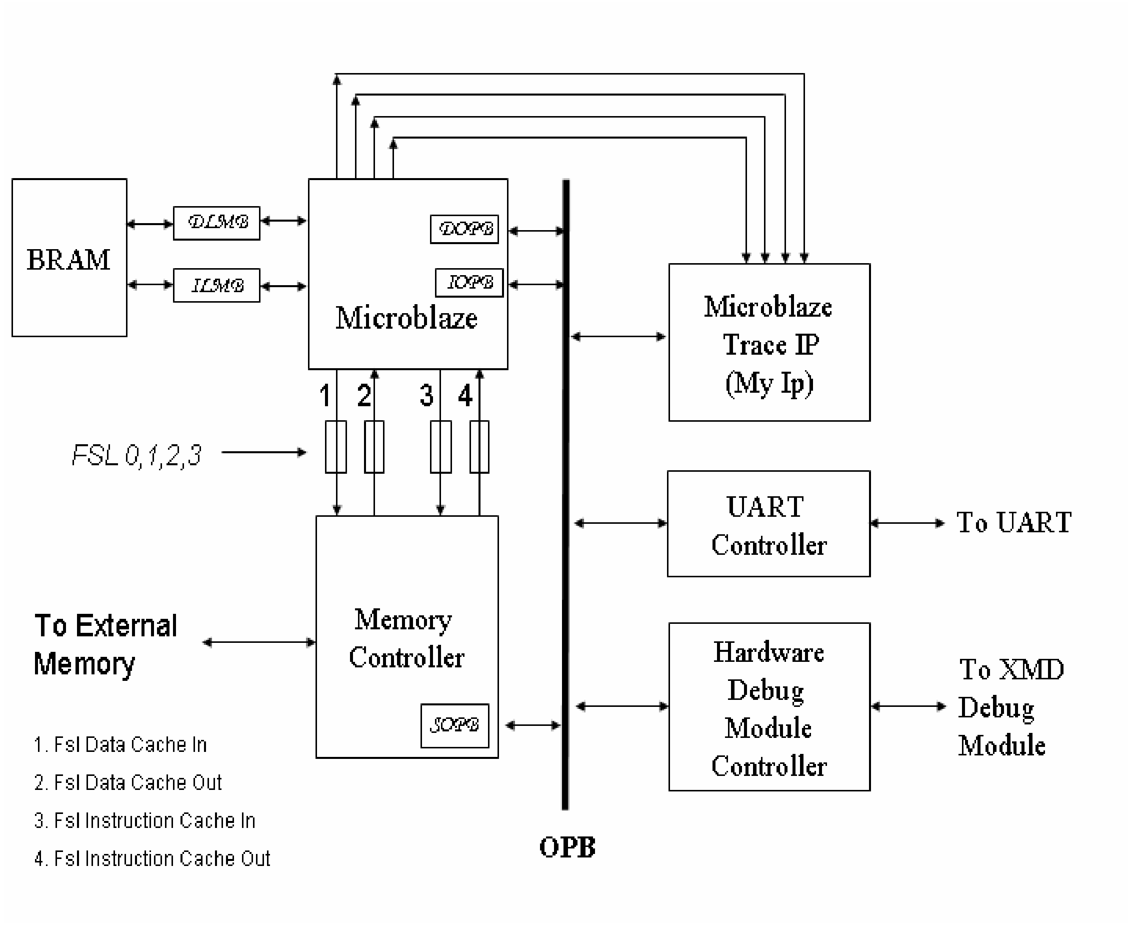


Figure 2.8 Microblaze design when caching is done using FSL Cache link.

In the previous two models instruction and data was transferred through OPB. In previous models OPB bus was shared among all the peripherals including Microblaze and memory controller. So the overhead of transfer of instruction and data was quite more.

In this model a dedicated Fast Simplex FSL link is established between Microblaze and Memory controller. This is simplex link therefore two links for data and two links for instructions are used for transferring data and instructions.. This model is not as good as previous one because FSL link does not perform better than Xilinx cache link.

Still this model is better than the first and second model because this uses dedicated link for instruction and data cache.

2.1.6 Summary of work

The above mentioned systems has been built using EDK tool [1] and new IP is configured on the OPB bus in order to keep track on the signals of Microblaze to find hit and miss for instructions and data.

For the new IP, a VHDL[4] code has been written and a new device driver[10] is written in C with the help of which Trace IP can communicate with the application program. Using this system different combinations of instructions and data cache are observed and a high performance gain is achieved while using both at the same time.

XMD hardware debug module is used which helps in debugging the system in order to transfer program into external memory.

GNU tool is used for compiling applications. EDK includes the GNU compiler tools for both the PowerPC and MicroBlaze processors. The MicroBlaze GNU tools include mb-gcc compiler, mb-as assembler and mb-ld loader/linker.

Whole system simulation is done in ModelSim which is very useful tool for tracking each and every the signal used in the system.

2.2 SystemC

2.2.1 System Overview

SystemC is a C++ class library and a methodology that you can use to effectively create a cycle-accurate model of software algorithms, hardware architecture, and interfaces of your SoC (System On a Chip) and system-level designs. We can use SystemC and standard C++ development tools to create a system-level model, quickly simulate to validate and optimize the design, explore various algorithms, and provide the hardware and software development team with an executable specification of the system. An executable specification is essentially a C++ program that exhibits the same behavior as the system when executed.

C or C++ are the language choice for software algorithm and interface specifications because they provide the control and data abstractions necessary to develop compact and efficient system descriptions. Most designers are familiar with these languages and the large number of development tools associated with them. **The SystemC Class Library provides the necessary constructs to model system architecture including hardware timing, concurrency, and reactive behavior that are missing in standard C++.** The C++ object-oriented programming language provides the ability to extend the language through classes, without adding new syntactic constructs. SystemC provides these necessary classes and allows designers to continue to use the familiar C++ language and development tools.

This design modeled is a based on the Xilinx based soft-core processor Microblaze and its associated OPB bus. A system is developed around this processor by attaching an IP for hardware acceleration and a synchronous External memory. The Processor can fetch instruction from the Instruction BRAM (Specific to FPGA) through the LMB or from the I-Cache if it is enabled. I-Cache is used only if the application is large enough to be stored in external memory. The peripherals are memory mapped through the OPB to the processor. There are separate ports and bus for data and instruction and the processor has Harvard architecture.

2.2.2 Cycle accurate timing Model

In this section the details of the actual Modeling of the system described above is discussed. Then the model is verified for timing using the actual implementation of the system in a FPAG and comparing the Cycles for a given application under different configurations.

2.2.3 Implementation

The three Steps for modeling the above system

1. The Design should be modular and communication should be separated from the functionality.
2. Interfaces are required for connecting the modules via ports.
3. The functionality and timing of individual module is modeled along with the channel implementation

The modules may be hierarchical, but at the top level the modules are

1. Microblaze
2. Instruction BRAM
3. Data BRAM

Since the model is hierarchical the Microblaze has sub modules they are :

1. I-Cache
2. D-Cache
3. Fetch
4. Decode
5. Execute.

Each of the modules is instantiated and port mapped through the interfaces. The transparency of each module is frozen at this stage of design, and each module is configurable at instantiation, thus allowing various configurations to be studied.

Configuration parameters for each module

1. MicroBlaze
    a. PC & MSR – Initial Program counter and MSR Value
2. Instruction BRAM
    a. Address – base address and high address
3. Data BRAM
    a. Address – base address and high address

At architectural level this is an important step of design.

2.2.4 Design Overview

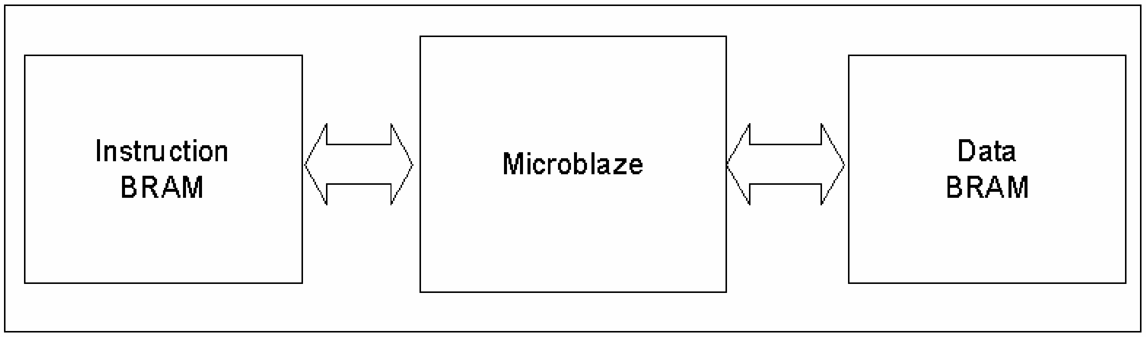


Figure 2.9 Top Level Design

In this design , the instructions are loaded in the Instruction BRAM from where it is fetched to microblaze for execution. The program counter (PC) in the Microblaze increments step by step in order to fetch next instruction. There is a Interface provided for the system in order to load the instructions. In this case i have loaded my instructions in the “inst.txt” file from where the design gets started. The instruction gets fetched to the microblaze and in case of storing of any local data, it gets stored in the Data BRAM. In this case I have kept my Data BRAM as an integrated array in the program. The size of the Instruction and Data BRAM can be configured. This configuration can be done at the time of initialization.

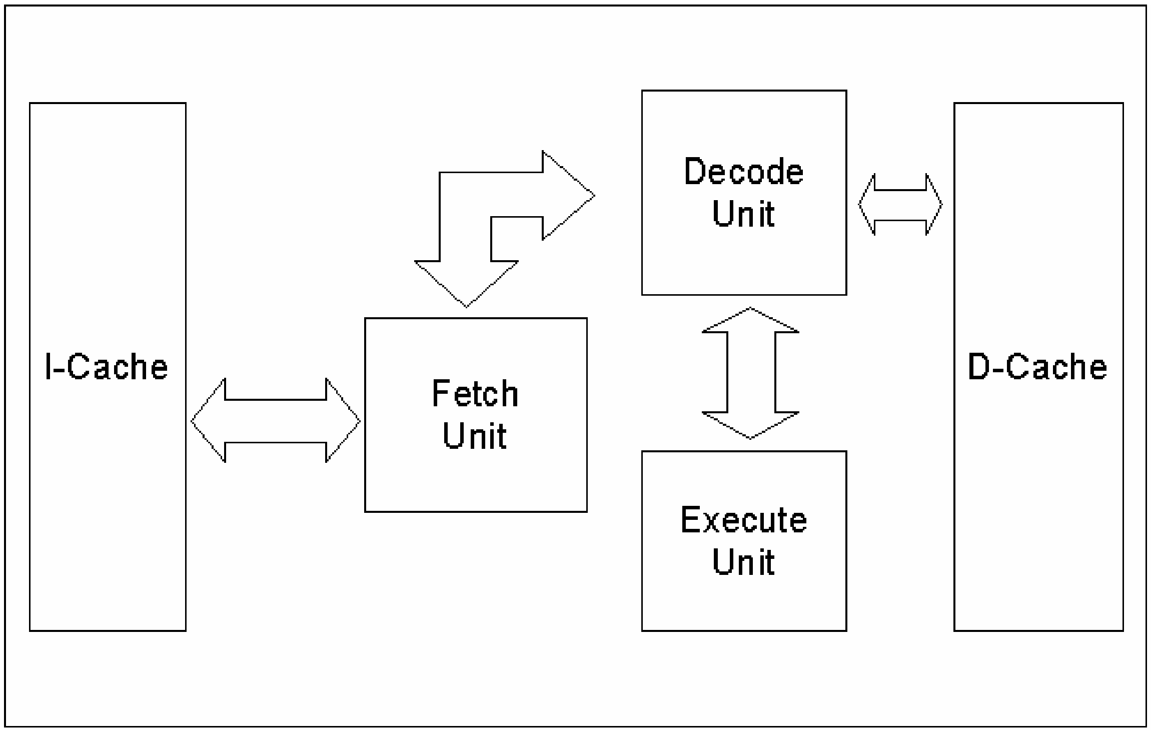


Figure 2.10 Microblaze Internal Module

This is the module which is made inside Microblaze. Here there are two different caches. First one is I-Cache and second one is D-Cache. The I-Cache is responsible for fetching the instructions from the external Instruction BRAM which is visible to the microblaze but as the signals are integrated in the Microblaze, they are also connected to the external Instruction BRAM through Microblaze. The fetch unit is responsible for fetching of instructions and where it gives the data to decode unit. The decode unit is decoded the instructions and if there is any need of storing or loading of local data then it is done in the D-Cache which is in turn again connected to the external Data BRAM. While decoding appropriate delay is provided in each instruction opcode so that proper timing model can be achieved.

2.3 PowerPC

The PowerPC central-processing unit (CPU) implements a 5-stage instruction pipeline consisting of fetch, decode, execute, write-back, and load write-back stages. The fetch and decode logic sends a steady flow of instructions to the execute unit. All instructions are decoded before they are forwarded to the execute unit. Instructions are queued in the fetch queue if execution stalls.

The fetch queue consists of three elements:
Two prefetch buffers
and a decode buffer.

If the prefetch buffers are empty instructions flow directly to the decode buffer. Up to two branches are processed simultaneously by the fetch and decode logic. If a branch cannot be resolved prior to execution, the fetch and decode logic predicts how that branch is resolved, causing the processor to speculatively fetch instructions from the predicted path. Branches with negative-address displacements are predicted as taken, as are branches that do not test the condition register or count register. The default prediction can be overridden by software at assembly or compile time.

Instruction and Data Caches

The PPC405 accesses memory through the instruction-cache unit (ICU) and data-cache unit (DCU). Each cache unit includes a PLB-master interface, cache arrays, and a cache controller. Hits into the instruction cache and data cache appear to the CPU as single-cycle memory accesses. Cache misses are handled as requests over the PLB bus to another PLB device, such as an external-memory controller.
The PPC405 implements separate instruction-cache and data-cache arrays. Each is 16 KB in size, is two-way set-associative, and operates using 8 word (32 byte) cache lines. The caches are non-blocking, allowing the PPC405 to overlap instruction execution with

reads over the PLB (when cache misses occur). The cache controllers replace cache lines according to a least-recently used (LRU) replacement policy. When a cache line fill occurs, the most-recently accessed line in the cache set is retained and the other line is replaced. The cache controller updates the LRU during a cache line fill. The ICU supplies up to two instructions every cycle to the fetch and decode unit. The ICU can also forward instructions to the fetch and decode unit during a cache line fill, minimizing execution stalls caused by instruction-cache misses. When the ICU is accessed, four instructions are read from the appropriate cache line and placed temporarily in a line buffer. Subsequent ICU accesses check this line buffer for the requested instruction prior to accessing the cache array. This allows the ICU cache array to be accessed as little as once every four instructions, significantly reducing ICU power consumption. The DCU can independently process load/store operations and cache-control instructions.

2.3.1 Cache Organization

The PPC405 contains an instruction-cache unit and a data-cache unit. Each cache unit contains a 16 KB, 2-way set-associative cache array, plus control logic for managing cache accesses. The caches contain copies of the most frequently used instructions and data and can typically be accessed much faster than system memory.

Each cache array is organized as a collection of *cachelines*. There are a total of 512 cachelines in a cache array, divided evenly into two *ways* (one way contains 256 lines). Line *n* from way A and line *n* from way B make up a *set* of cachelines, also known as a *congruence class*. A cache array contains a total of 256 sets, or congruence classes.
Each cacheline contains the following pieces of information:

• A *tag* used to uniquely identify the line within the congruence class.
• 32 bytes of *data* that are a copy of a contiguous, 32-byte block of system memory, aligned on a 32-byte address boundary. The data can represent either instructions (in the instruction cache) or operands (in the data cache).

• An*LRU* bit that specifies which cacheline within the congruence class is least-recently used. Each time a cacheline is accessed, the cache controller marks the *other* line within that congruence class as least-recently used. When a new cacheline is read from memory during a cacheline fill, the line in the congruence class marked least recently used is replaced.

• A*dirty* bit that indicates whether the cacheline contains modified information. A modified cacheline contains data that is more recent than the copy in system memory.

The instruction cache does not have a dirty bit.
The 512 total lines of 32 bytes each yields a 16 KB cache size.

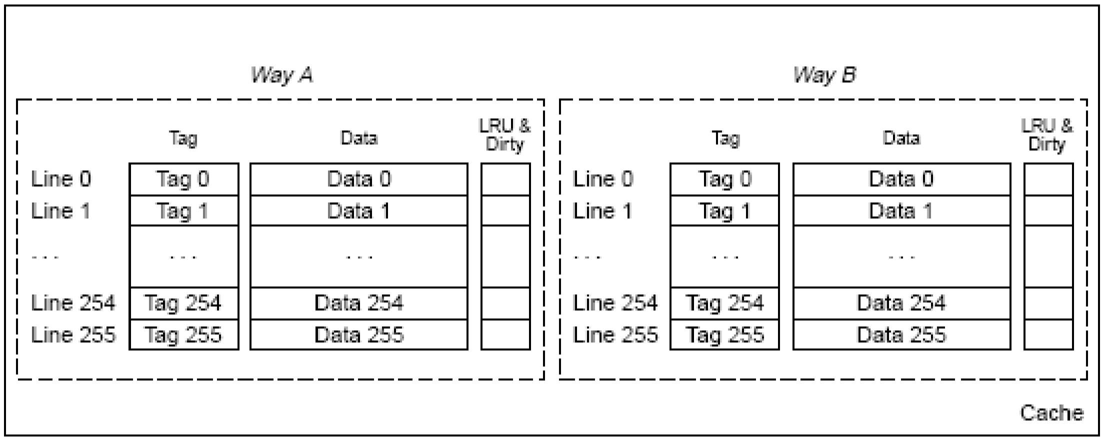


Figure 2.11 Logical Structure of cache array for PowerPC

Data is selected from the data cache using fields within the data address. Likewise, an instruction is selected from the instruction cache using fields within the instruction address. The data cache is *physically tagged* and *physically indexed*. This means that the physical address alone is used to access the data-cache array. The instruction cache is *physically tagged* and *virtually indexed*. Here, the effective address is used to specify a congruence class (set of lines) within the cache, and the physical address is used to specify a specific tag. The instruction cache is accessed in this manner for performance reasons.

The line field in the data address is used to select a congruence class from the cache array. The congruence class contains two lines, one from each way. Each line contains a tag, meaning two tags are present in a congruence class. The tag field in the data address is compared to both tags in the congruence class. A *hit* occurs when the data address tag field is equal to one of the two tags. A *miss* occurs when the data-address tag field is not equal to either of the tags.

2.3.2 Instruction-Cache Operation

Instructions flow from the instruction-cache unit (ICU) to the execution pipeline.

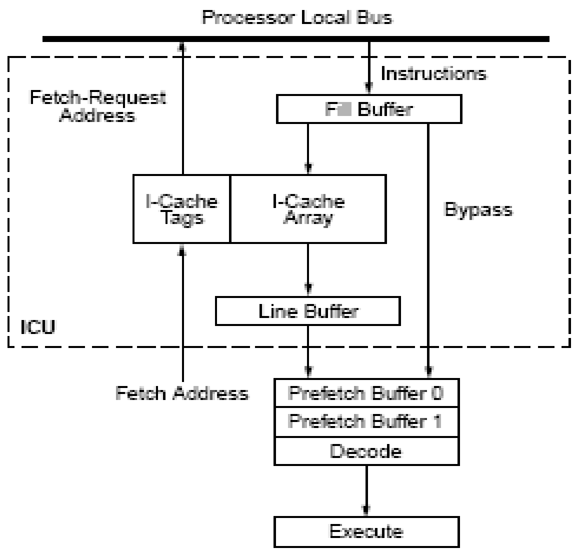


Figure 2.12 Instruction flow from instruction cache unit.

All instruction-fetch requests are handled by the ICU. If a fetch address is cacheable, the ICU examines the instruction cache for a hit. When a hit occurs, the cacheline is read from the instruction cache and loaded into the line buffer. Individual instructions are sent from the line buffer to the instruction queue. From there they are either loaded into one of the prefetch buffers or are immediately decoded, depending on the current state of the decode and execution pipelines. Up to two instructions per clock cycle can be sent to the instruction queue from the line buffer.

When a cache miss occurs, or when an instruction address is not cacheable, the ICU sends the fetch-address request to system memory over the processor local bus (PLB). A

cache miss results in a cacheline fill, which appears as an eight-word request on the PLB. The request size for non-cacheable instructions can be either four words (half line) or eight words (full line) and is programmable using the CCR0 register. Full-line (cacheable and non-cacheable) and half-line fetch requests are always completed (never aborted), even if the instruction stream branches before the remaining instructions are received. As instructions are received by the ICU from the PLB, they are placed in the fill buffer.
The ICU requests the target instruction first, but the order instructions are returned depends on the design of the PLB device that handles the request (typically a memory controller). When the ICU receives the target instruction, it is immediately forwarded from the fill buffer to the instruction queue over the bypass path. The remaining instructions are received from the PLB and placed in the fill buffer. Subsequent instruction fetches read an instruction from the fill buffer if it is already present in the buffer. If a cache miss occurred, the instruction-cacheline is loaded with the fill-buffer contents after all instructions are received.

2.3.3 Data-Cache Operation

Data flows between the data-cache unit (DCU) and the general purpose registers.

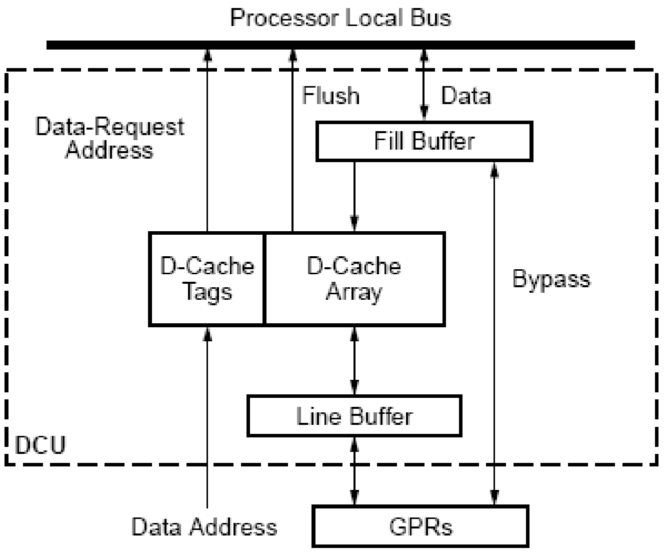


Figure 2.13 Data flow to from data cache unit

All data-load requests and data-store requests are handled by the DCU. If a data address is cacheable, the DCU examines the data cache for a hit. A hit causes the cacheline to be read from the data cache and loaded into the line buffer. For a load hit, the data value is read from the line buffer and written to a GPR. For a store hit, the data value is read from the GPR and written to the line buffer and the line buffer is stored back into the data cache. The data cache supports byte writeability to improve the performance of byte and halfword stores. Load hits and store hits can be completed in one clock cycle.

If a cache miss occurs or if the data address is not cacheable, the DCU sends the data address request to system memory over the processor local bus (PLB). Store misses to write-back memory and all load misses cause a cacheline fill. The size of all cacheline fill requests over the PLB is 32 bytes. The request size for a store to write-through memory (cache hit and cache miss) is one word (four bytes). The request size for a non-cacheable data access is programmable using the CCR0. Cacheline fills are always completed (never aborted) even if the processor does not require any other bytes in the line. As data is received by the DCU from the PLB, it is placed in the fill buffer.

During a cacheline fill, the DCU requests the target data (load or store) first. However, the order data is returned depends on the design of the PLB device that handles the request (typically a memory controller). The remaining data is received from the PLB and placed in the fill buffer. Subsequent loads and stores access the fill buffer if the data is present in the buffer. The data cacheline is loaded with the fill-buffer contents after all data are received.

2.3.4 Image Processing application

2.3.4.1 System overview

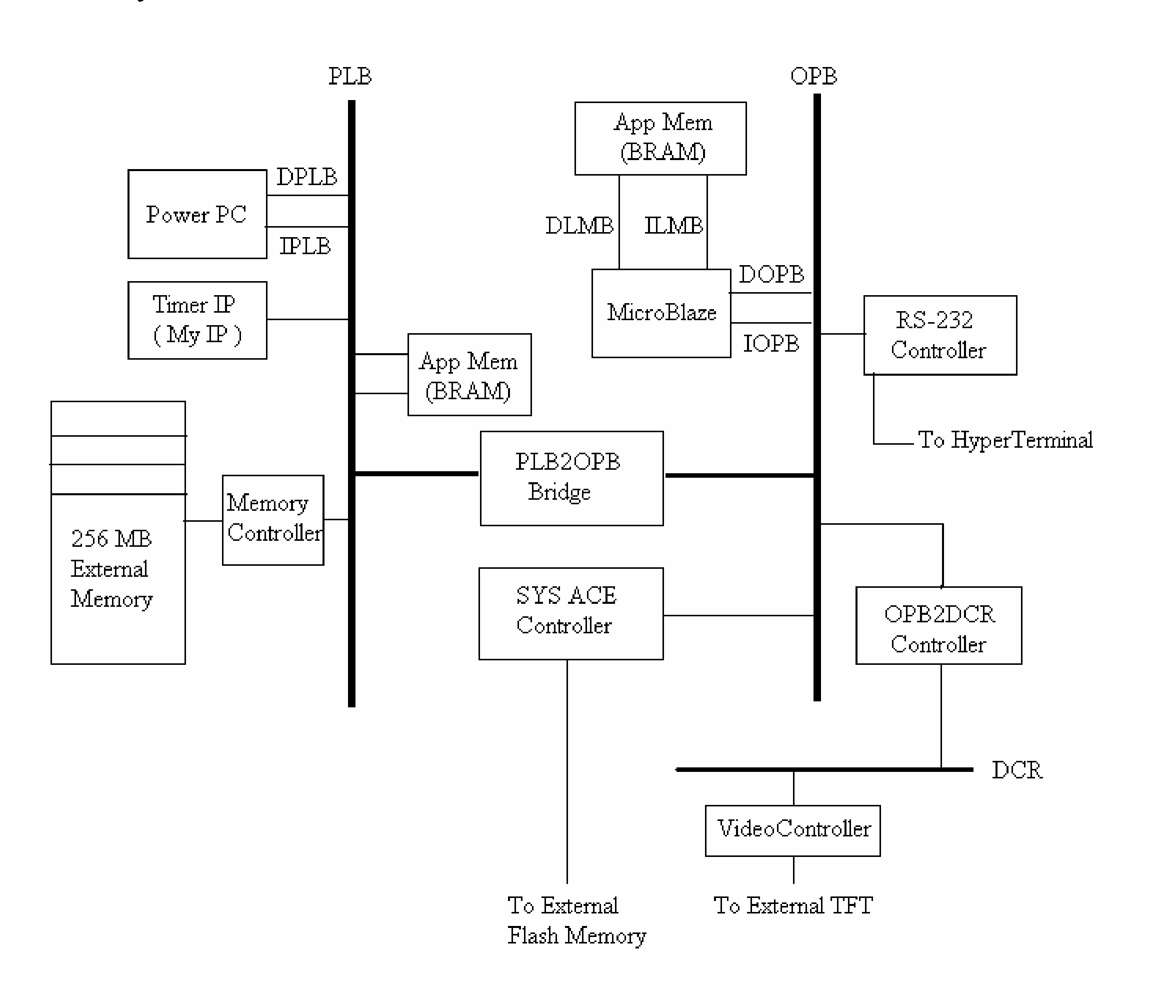


Figure 2.14 System View for Image Processing application

Application Flow

- Images are loaded in the Flash memory using flash writer.
- The Program is loaded on the board using USB cable
- Application is loaded in the External RAM.
- Images are uploaded form flash memory to the external RAM
- PowerPC takes care of loading data from flash to external memory
- Application executes in the PowerPC.
- Cache is enabled for PowerPC.
- Instructions and Data is cached for faster execution.
- Image file is decoded and processed in PowerPC
- The final RGB pixels are drawn on the TFT through

PLB -> PLB2OPB Bridge

PLB2OPB Bridge -> OPB

OPB -> OPB2DCR Bridge

OPB2DCR Bridge -> DCR

DCR -> Video controller

Video Controller -> TFT

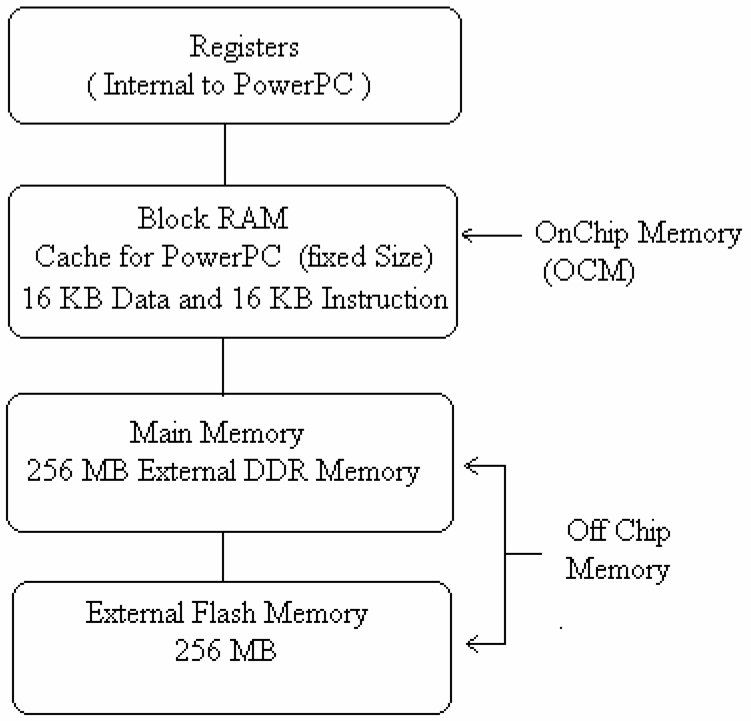


Figure 2.15 Memory Hierarchy

Image processing Application

In this application image is loaded from the flash memory to the 256 External DDR RAM from where it is being processed in different forms. Basically I have implemented three type of image processing

1. Revert Image : In this type the image colours are reverted for example Black is converted to white , white is converted to black, blue to orange , orange to blue and so on.
2. Mirror Image : In this type the image is mirrored so that left side goes to right and right side moves to left.
3. Mixed Image : This is a combination of both Mirror and Revert image.

This file is stored in a per pixel format so we can change the value of each pixel in order to alter the file there are basically three colors used in the file they are Red, Blue and Green . These colors can be changed as we knot that the value of any color lies in the range of 0 - 255. Applying certain computations on these pixel we ca change the image. The image application opens a file that is stored in a flash memory. This file is in a predefined format and needs to be opened in a specific format. There are basically two header parts of the file. The first part describes details about file whereas second part defines the detailed info about content of file. These two headers are as follows

1) BITMAPFILEHEADER
2) BITMAPINFOHEADER

A color table is an important criterion for image processing as it contains the details of each pixel value. This image contains 32 bit color value for storing a pixel. The colors which are mainly used for this purpose are Red , Green and Blue. Each color takes 8 bit value so we can modify the value of color on the basis of value ranging from 0-255. This value of a color shows its intensity. The rest 8 bits are unused. The example of color fields table which is stored at various locations in the memory is as follows:

ColorTable

Address	Blue	Green	Red	Unused
[00000000]	84	252	84	0
[00000001]	252	252	84	0
[00000002]	84	84	252	0
[00000003]	252	84	252	0
[00000004]	84	252	252	0
[00000005]	252	252	252	0

|
|
Other image Data

Since we know that the external memory is not portioned into parts so in order to save image we have to explicitly portion the external memory into logical parts in order to save the image so that they must not overlap on each other. The External memory is logically portioned in fixed memory of 2MB and in each partition of this memory one image is copied and then it is executed from that portion of memory. The logical portioning of memory is given as follows

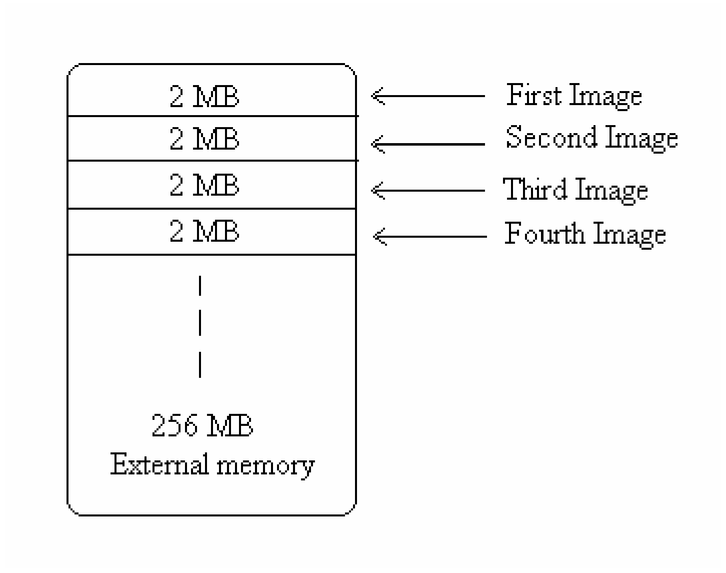


Figure 2.16 Logical portioning of External memory

2.3.4.2 Attaching a new IP for PLB

There is an internal clock provided in the PowerPC but due to certain reasons it is not working properly so I need to attach a new Timer IP to the system. The Timer IP is provided for the OPB bus but not for PLB bus.

Following are the steps for attaching a new IP to the system.

1. Go to add new peripheral and make a new IP for the system.
2. Select the bus, on which this IP will reside, it can be either PLB or OPB.
3. Once the bus is selected we need to select the number of resisters required in this IP (Reg no. = 1 in Timer IP). This register seems to be a signal for VHDL code whereas it seems to be a Software register for a C code. This is the main point where hardware and software can communicate.

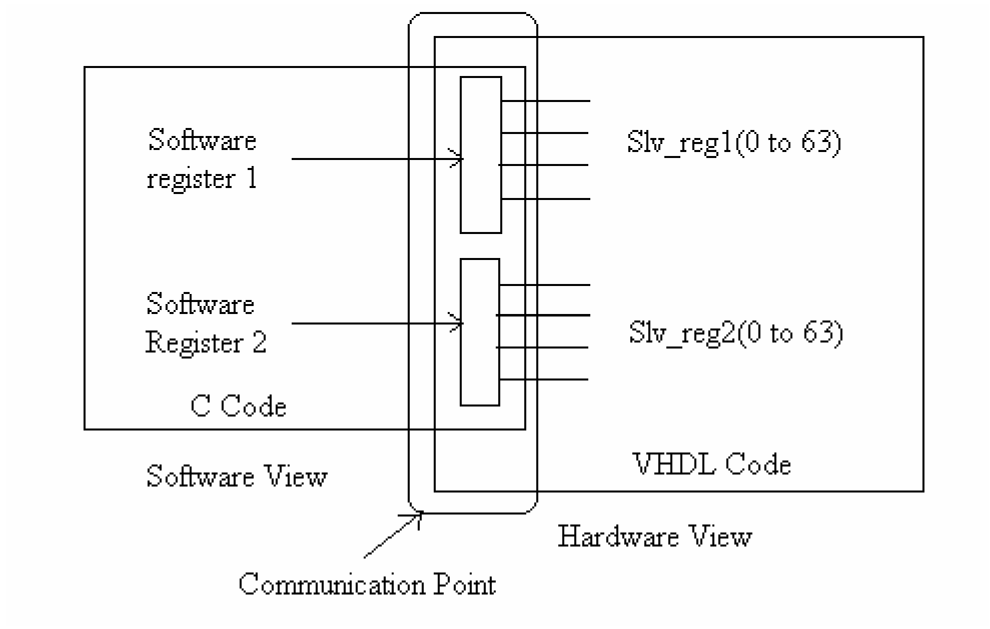
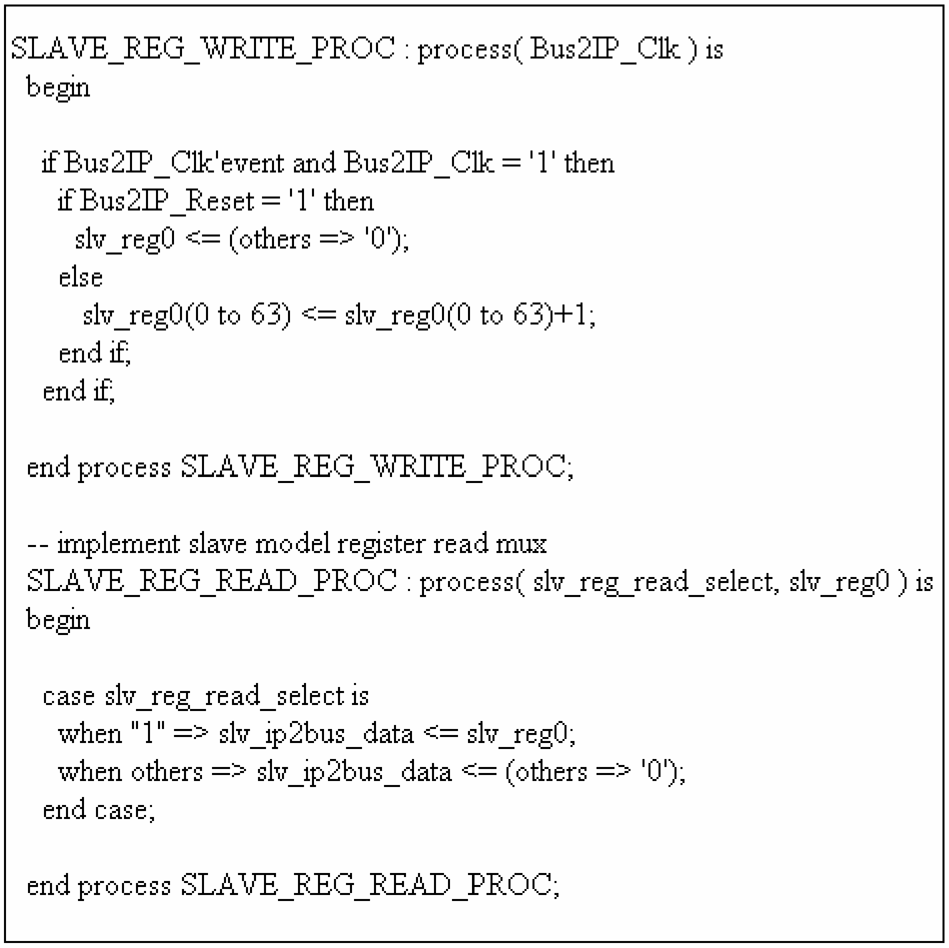


Figure 2.17 Hardware and Software view of a Register

4. After the number of registers is selected, a new IP is created for the PLB/OPB bus. Now we need to configure this new IP according to our need.
5. There are two files provided in the Pcore IP. Select the user_logic.vhdl file and add the following code in it.



```
SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
 begin

   if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
     if Bus2IP_Reset = '1' then
       slv_reg0 <= (others => '0');
     else
        slv_reg0(0 to 63) <= slv_reg0(0 to 63)+1;
     end if;
   end if;

 end process SLAVE_REG_WRITE_PROC;

 -- implement slave model register read mux
 SLAVE_REG_READ_PROC : process( slv_reg_read_select, slv_reg0 ) is
 begin

   case slv_reg_read_select is
     when "1" => slv_ip2bus_data <= slv_reg0;
     when others => slv_ip2bus_data <= (others => '0');
   end case;

 end process SLAVE_REG_READ_PROC;
```

This file can be tested in various simulation models. ModelSim is used for testing of this IP. This works perfectly well. For each rising edge of the clock it increments the value of timer by 1. Hence at any given point of time we can check the value of timer using the IP device driver in the application.

How to add IP to the system

1. This IP should be kept in Pcores folder of project.
2. Its name gets added in the Project repository sub list. Once this is added we can add it to the system. But this is not all done. We must add IP to the bus and add signals and map address of the IP.
3. IP can be accessed through a unique address on the bus so in order to assign address to the IP we need to add address in the address space. It should be noted that this address does not conflict with any other IP of the system.

4. This module is created for PLB bus so just add it to PLB Bus as a slave.
5. Last thing to be done is giving a clock and a reset signal to the IP. The common clock signal for this system is sys_clk_s. For reset it should be mapped with sys_rst_s.

After the IP is attached we have to write a device driver code for the application. A device driver is a code written in C/ C++ which helps our application to communicate with hardware. A device driver, or software driver is any computer program that allows other programs to interact with a computer hardware device, or else to work as if they are interacting with a particular device. In other words, a driver is an interface for communicating with the device, or emulates a device. A driver typically communicates with the device through the computer bus or communications subsystem that the hardware is connected to. When a program invokes a routine in the driver, the driver issues commands to the device, and when the device sends data, the driver invokes routines in the program.

Drivers are hardware-dependent and operating-system specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interfacing needs.

```
/* the device component data type */
typedef struct
{
Xuint32 BaseAddress; /* component data variables */
Xuint32 IsReady;
Xuint32 IsStarted;
} XDevice;
/* create an instance of a device */
XDevice DeviceInstance;
/* device component interfaces */
XStatus XDevice_Initialize(XDevice *InstancePtr, Xuint16 DeviceId);
XStatus XDevice_Start(XDevice *InstancePtr);
```

The value of Device ID , Base address and High Address is available in xparameter.h file. Looking at this code one can easily configure the device for customized use. In my application I have integrated my device driver code in application. By using a function " XIO_In32(Timer base address); "

The system is well tested for the image processing application with and without cache. There are basically four typed of caching of application which gives rise to the increase in computation speed of application. These are

1. Without using Instruction and Data Cache
2. With Instruction Cache but without Data Cache.
3. With Data Cache but without Instruction Cache.
4. With both Instruction and Data Cache.

We know that the instructions are frequently used data by the processor so when instructions are cached, we get a high performance gain in speedup. But in case of this application it has mainly to deal with data part as the image is nothing but huge data. So the gain factor in speed is quite high when we cache data in our application.

3. RESULTS

3.1 Microblaze

When Only Data is cached

Matrix Multiplication (XxY)	D-cache is On	text	data	bss
5x5	33385	3344	745	32795
50x50	25874788	3492	745	32791

When only Instructions are cached

Matrix Multiplication (XxY)	I-Cache is On	text	data	bss
5x5	7416	3344	745	32795
50x50	6620705	3344	745	32791

When Both instructions and data are cached

Matrix Multiplication (XxY)	D-cache and I-cache is On	text	data	bss
5x5	2003	3492	745	32795
50x50	1687005	3492	745	32791

When Both instructions and data are not cached

Matrix Multiplication (XxY)	D-Cache and I-Cache is Off	text	data	bss
5x5	37512	3196	745	32795
50x50	30091340	3196	745	32791

These are some of the results which were taken on the system which is described on page no. 27. Various applications with different size of matrix were tested on the system using different configuration of cache. The size of application was kept constant this is shown in table in text, data and bss column. Since the size of application is constant the only difference is size of data which is created at run time. Various size of matrix was tested on this system and a graph was plotted. The result shows that as compared to data, more performance is gained in instruction. This is because instructions are more frequently used in this application. DXCL link form Microblaze is not used very frequently.

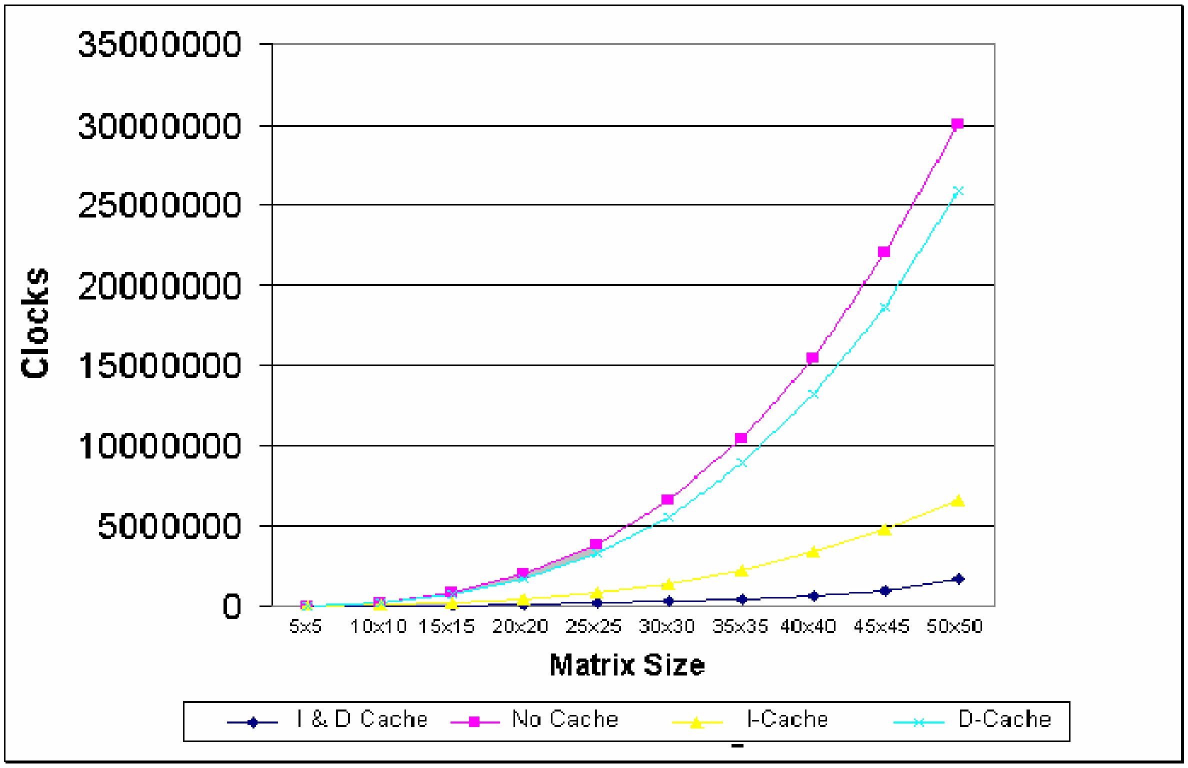


Figure 3.1 Comparative results for different configuration of caches for Microblaze.

This graph is a drawn at various size of matrix. In this application the size of cache is kept constant which is 16KB for Instruction and 16 KB for Data. The graph clearly shows that as we go on increasing the size of matrix we get high performance gain while caching instructions as compared to data. This is because instructions are executed more frequently as compared to data in the application.

OUTPUT OF PROGRAM

When simple Multiplication program

Runs in BRAM

Number of clock cycles required is 715

Runs in External RAM

Number of clock cycles required is 13671

Runs in External memory when cache is enabled

Number of clock cycles required is 1657

Output for 50 x 50 matrix multiplication when read miss, Data write hit and data write miss is tracked.

Both instructions and data are cached at 16Kb

Number of clock cycles required is is 1685560

No Of Data Read Misses :: 11723

No Of Instruction Misses :: 85

Output for 100 x 100 matrix multiplication when read miss, Data write hit and data write miss is tracked.

Number of clock cycles required is 18994847

No Of Data Read Misses :: 267703

No Of Instruction Misses :: 85

When we use the instruction "Use_Icache" and "Use_Dcache" in C then the ASM code of both is same for three instructions so there is gain in performance even if we write a code of addition for two numbers because instructions is cached at this point.

Moreover when we apply different combination of matrix multiplication only content of registers of r18 and r19 is getting changed. So the number of Instruction miss is constant in this case. Data miss is continuously increasing in this case because the data fetched for Matrix Array is called up for number of times and it is continuously getting miss in cache.

3.2 SystemC

30 instructions are modeled properly with a clock delay of single cycle. Some of the instructions are as follows.

add R1, R2, R3

add R1, R2, R3 with carry

add R1, R2, R3 with carry and keep carry

addi R1, R2, #value

AND R1,R2,R3

andi R1, R2, #value

ANDnot R1,R2,R3

bne R1, R2, label

sw R1, R2, offset.

…..

…..

This design is not only a cycle approximate model but also stimulates the actual microblaze design. This is more or less a behavior model which is used to study and examine the working of Microblaze.

In this design the time required for the computation of each single cycle instruction is 10ms. Therefore for a single clock cycle 10ms is needed.

So 1 Clock cycle = 10 ns for this design.

also we can say that the speed is $1/ (10 * 10^{-9}) = 10^{8}$ cycles per second = 100 MHz.

This is the frequency of Microblaze used in the system.

The whole Microblaze has been modeled. The instructions can be fetched from a file which is stored in "inst.txt" and data can be stored locally in the array part. Moreover all the local register of Microblaze can be configured as per requirement. MSR is checked and at any time the status of all the registers can be viewed. At the execution time of the program the MSR and all local registers are updated so its value can be viewed after each instruction. The total time requirement of the design at each instruction is carried with a local clock which is modeled in the design.

This design basically performs three main task :

1. Viewing of contents of local registers at any point of instruction.

2..Calculating time required for computation of instructions.

3. Studying the simulation of Microblaze.

3.3 PowerPC

Output of Image processing Application on PowerPC

```
Instruction cache is ON
Data Cache is ON

Reading file : 1
Type of file is : 19778
Size of file is : 983094
reserved1 of file is : 0
reserved2 of file is : 0
offset of file is : 54
File size is 983094 bytes
Offset to image data is 54 bytes
Image size = 640 x 512
Number of color planes is 1
Bits per pixel is 24

Reading file : 2
Type of file is : 19778
Size of file is : 983094
reserved1 of file is : 0
reserved2 of file is : 0
offset of file is : 54
File size is 983094 bytes
Offset to image data is 54 bytes3094
Image size = 640 x 512
Number of color planes is 1
Bits per pixel is 24

………..
………..
…………
…………
…………
…………

Time required for computation: 1669324 cycles.
```

Comparison of results

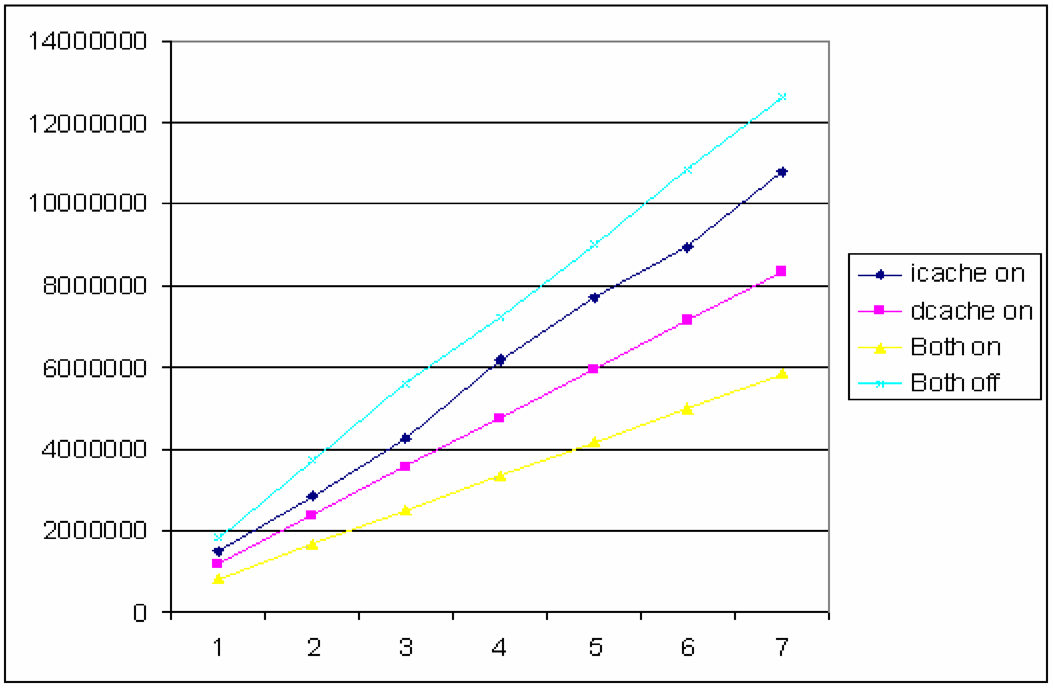


Figure 3.2 Comparative result for different configuration of caches for PowerPC

Tabular form

No. of Images	Both Cache off	I-cache ON	Gain Ratio I-Cache	D-Cache ON	Gain Ratio D-cache	Both I and D ON	Gain Ratio I & D cache
1	1809998	1500337	1.2	1197707	1.51	836977	2.16
2	3729994	2834520	1.31	2391502	1.55	1669324	2.24
3	5594344	4249072	1.31	3584422	1.56	2501056	2.23
4	7213484	6175721	1.17	4778200	1.51	3333481	2.16
5	9016292	7718060	1.16	5971158	1.50	4165239	2.16
6	10820264	8928029	1.21	7165016	1.51	4997595	2.16
7	12623647	10804383	1.16	8358387	1.51	5829637	2.16

This result shows that the relative comparison of computational time when image is loaded and processed. The results were obtained while loading different number of images on the system and with different configuration of cache. The graph shows high performance when both instruction and data is cached. But in this case D-Cache shows more speedup than just I-Cache. The reason is that the application has mainly to deal with data as compared to instructions. The table shows the gain ratio while performing I and D cache.

When only I is cached the gain ratio is approximately 1.2

When D is cached the gain ratio is approximately 1.5

When I and D are cached the gain ratio is approximately 2.16.

4. Conclusion

User requires a large amount of memory to store and access the data in a program, for this purpose on chip Block RAM is available but due to limited size of OnChip Block RAM we are forced to store and run our application using Offchip memory (External Memory). But due to latency of access of instructions and data on Offchip memory, our application shows worse performance as we go on increasing the size of program.

So in order to increase the performance gain in our application we have used Cache so that instructions and data which are frequently used is stored on the Onchip memory and when required it is accessed from Onchip memory instead of using data from slower device.

Our result shows that when we cache instructions and data we get a speedup factor which is very much higher than without using cache. Speed up is more if instructions are cached and data is not cached (for simple programs like matrix multiplication). On the other hand if only data is cached while instruction is not cached then the speedup is high (in case of application like image processing in which data processing is quite high). So it is always important to cache the instructions as they are frequently used.

We have also seen that even if the cache is ON then there is some compulsory misses for instructions and data. On the other hand if our program resides completely in BRAM then there is no compulsory miss. Result shows the same fact.

5. Future scope

This project is basically divided in two parts

1. Development of Microblaze and PowerPC based system in EDK.
2. SystemC modeling of the system.

In Microblaze no Memory Management unit is available for the system, so one can develop MMU (Memory Management Unit) so that the storing and loading of programs can be very well partitioned. The External memory is been logically portioned in 2MB memory of equal size but if memory management unit takes care of it then a lot of space in external memory can be saved. Also one can add multi processors on this system and can execute whole application parallel which will give rise to more performance gain.

Where as in SystemC Part, only a few part of the system has been modeled. But the further enhancement of the project will be quite easy, as the Microblaze has been properly modeled. The instructions, clock cycles and pseudo code for the Microblaze and PowerPC is given in their document. So a fully functional model for Microblaze and PowerPc can itself be a project which is feasible. After modeling of Microblaze , OPB bus with other peripherals can be modeled.

Acronyms

BRAM	: Block Random access Memory.
DLMB	: Data Local Memory Bus
DOPB	: Data Onchip Peripheral Bus.
DSOCM	: Data side Onchip peripheral bus.
DXCL	: Data Xilinx cache link.
EDK	: Embedded development Kit
FSL	: Fast simplex link.
ICU	: Instruction Cache Unit.
ILMB	: Instruction Local Memory Bus.
IOPB	: Instruction Onchip Peripheral Bus.
IP	: Intellectual property.
ISOCM	: Instruction side OnChip peripheral bus.
IXCL	: Instruction Xilinx cache link.
MCH	: Memory channel.
OCM	: On chip memory.
OPB	: Onchip peripheral bus.
PLB	: Processor Local Bus.
SOPB	: Slave OnChip Peripheral Bus.
XMD	: Xilinx Debug Module.

References

[1] Embedded Development Kit, Getting Started with EDK, 1-800-255-7778 EDK 7.1i February 15, 2005 http://www.xilinx.com

[2] MicroBlaze Processor Reference Guide,1-800-255-7778 UG081 (v5.0) January 20, 2005 http://www.xilinx.com

[3] FPGA Field Programmable Gate Array, http://www.andraka.com/whatisan.htm

[4] VHDL Very High Speed Integrated Circuit Hardware Description Language http://ghdl.free.fr/ghdl/index.html

[5] Fast Simplex Link FSL, DS449 December 1, 2005 www.xilinx.com

[6] MCH controllerMulti-CHannel (MCH) On-chip Peripheral Bus controller. DS496 July 1, 2005 www.xilinx.com

[7] PowerPC Processor. EDK 6.1 September 2, 2003 www.xilinx.com

[8] Relocating Data and code for embedded systems, XAAP642 (v1.0) October 21, 2002 www.xilinx.com.

[9] Platform studio User guide for EDK 6.2i UG113 (v1.0) March 12, 2004

[10] Device Driver Programmer Guide. v1.2 - July 31, 2002 www.xilinx.com

[11] XUP Virtex-II Pro Development System Hardware Reference Manual, Document Version: 0.00Document Date: August 2004