Intel® PXA27x Processor Family

Optimization Guide

August, 2004

Order Number: **280004-002**

Contents

Figures

Tables

Revision History

Date	Revision	Description
3/12/04	-001	Initial release
7/29/04	-002	Updates to Chapter 3

Introduction 1

1.1 About This Document

This document is a guide to optimizing software, the operating system, and system configuration to best use the Intel® PXA27x Processor Family (PXA27x processor) feature set. Many of these techniques may have already been implemented in Intel's operating system (OS) board support packages (BSPs). If a design is based on an Intel BSP, it must use the latest revision.

This document assumes users are familiar with the documentation shown in Table 1-1.

Table 1-1. Related Documentation

Document Title	Order Number
Intel® PXA27x Processor Family Developer's Manual	280000
Intel® PXA270 Processor Family Design Guide	280001
Intel® PXA270 Processor Electrical, Mechanical, and Thermal Specificiations	280002
Intel® PXA27x Processor Family with Intel® Folded Stack-Chip Scale Packaging Electrical, Mechanical, and Thermal Specifications	280003
Intel XScale® Microarchitecture for the PXA27x processor and Intel® PXA27x Processor Family User's Manual	11465, 11466
Intel® PXA250 and PXA210 Application Processors Optimization Guide	278552
Intel XScale® Core Developer's Manual	273473
The Complete Guide to Intel® Wireless MMX™ Technology	278626
Programming with Intel® Wireless MMX™ technology: A Developer's Guide to Mobile Multimedia Applications	ISBN: 0974364916
Intel StrataFlash® Wireless Memory datasheet	251902

This guide is organized into these sections:

- Chapter 2, "Microarchitecture Overview" presents an overview of the Intel XScale® Microarchitecture and the Intel® Wireless MMX™ technology media co-processor.
- Chapter 3, "System Level Optimization" discusses configuration of the PXA27x processor to achieve optimal performance at the system level.
- Chapter 4, "Intel XScale® Microarchitecture and Intel® Wireless MMX™ Technology Optimization" discusses how to optimize software (mostly at the assembly programming level) to take advantage of the Intel XScale® Microarchitecture and Intel® Wireless MMX™ technology media co-processor.
- Chapter 5, "High Level Language Optimization" is a set of guidelines for C and C++ code developers to maximize the performance by making the best use of the system resources.
- Chapter 6, "Power Optimization" discusses the trade-offs between performance and power using the PXA27x processor.
- Appendix A, "Performance Checklist" is a set of guidelines for system level optimizations which allow for obtain greater performance when using the Intel XScale® Microarchitecture and the PXA27x processor.

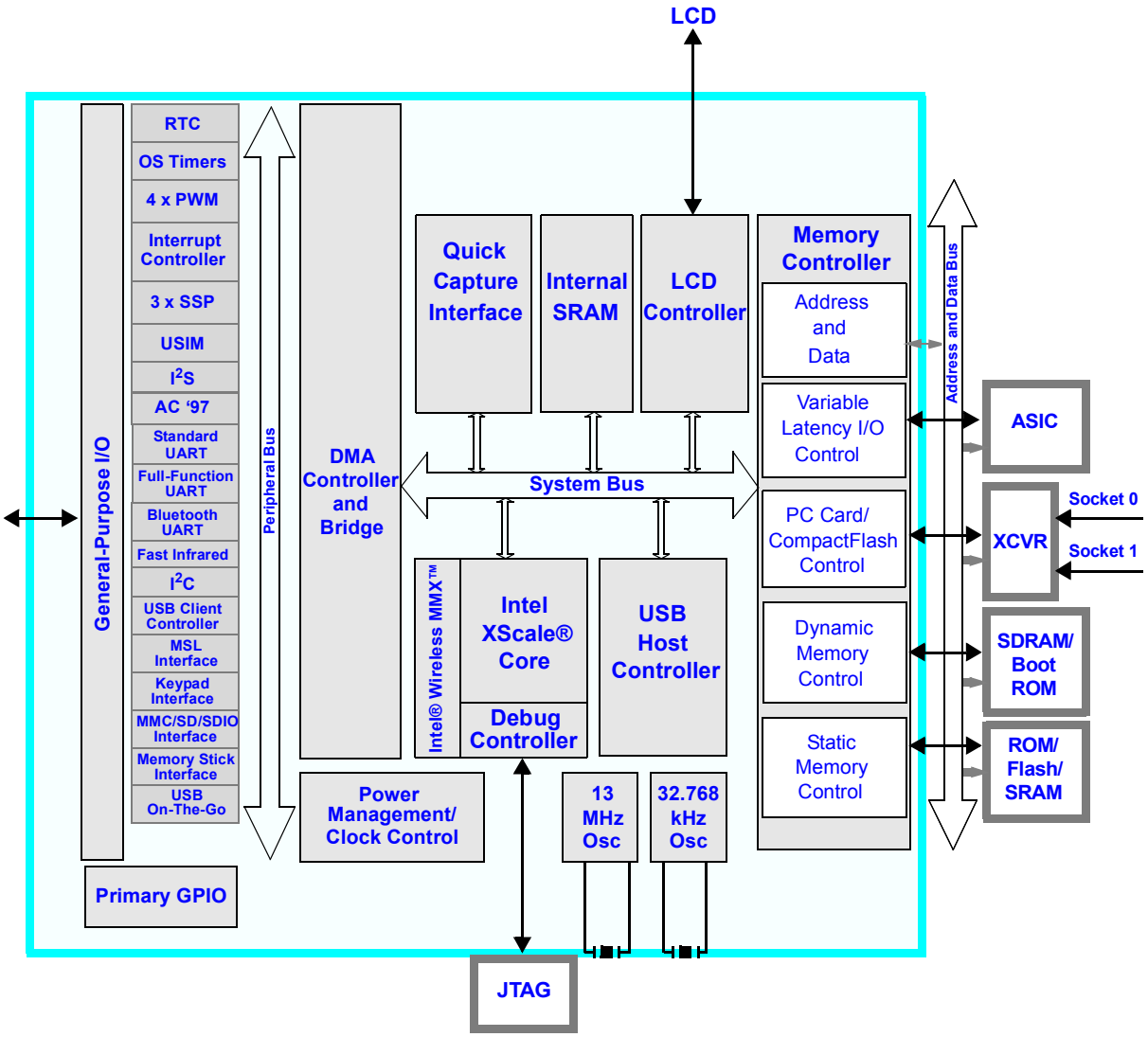
1.2 High-Level Overview

Mobile and wireless devices simplify our lives, keep us entertained, increase productivity and maximize our responsiveness. Enterprise and individual consumers alike realize the potential and are integrating these products at a rapid rate into their everyday life. Customer expectations exceed what is being delivered today. The desire to communicate and compute wirelessly—to have access to information anytime, anywhere—is the expectation. Manufacturers require technologies that deliver high performance, flexibility and robust functionality—all in the small-size, low-power framework of mobile handheld, battery-powered devices. The Intel® Personal Internet Client Architecture (Intel® PCA) processors with Intel XScale® Microarchitecture help drive wireless handheld device functionality to new heights to meet customer demand. Combining low-power, high-performance, compelling new features and second-generation memory stacking, Intel PCA processors help redefine what a mobile device can do to meet many of the performance demands of enterprise-class wireless computing and feature-hungry technology consumers.

Targeted at wireless handhelds and handsets such as cell phones and PDAs with full-featured operating systems, the Intel PXA27x processor family is the next generation of ultra low-power applications with industry-leading multimedia performance for wireless clients. The Intel PXA27x processor is a highly integrated solution that includes Wireless Intel Speedstep® technology for ultra low power, Intel® Wireless MMX™ technology, and up to 624 MHz for advanced multimedia capabilites, and Intel® Quick Capture Interface to allow customers to capture high-quality images and video.

The PXA27x processor incorporates a comprehensive set of system and peripheral functions that make it useful in a variety of low-power applications. The block diagram in Figure 1-1 illustrates the PXA27x processor “system on a chip”, and shows a primary system bus with the Intel XScale® Microarchitecture core (Intel XScale® core) attached along with an LCD controller, USB host controller and 256 Kbytes of internal memory. The system bus is connected to a memory controller to allow communication to a variety of external memory or companion-chip devices, and it is also connected to a DMA/bridge to allow communication with the on-chip peripherals. The key features of all the sub-blocks are described in this section, with more detail provided in subsequent sections.

Figure 1-1. PXA27x Processor Block Diagram



1.2.1 Intel XScale® Microarchitecture and Intel XScale® core

The Intel XScale® Microarchitecture is based on a core that is ARM* version 5TE compliant. The microarchitecture surrounds the core with instruction and data memory management units; instruction, data, and mini-data caches; write, fill, pend, and branch-target buffers; power management, performance monitoring, debug, and JTAG units; coprocessor interface; 32K caches; MMUs; BTB; MAC coprocessor; and a core memory bus.

The Intel XScale® Microarchitecture can be combined with peripherals to provide application-specific standard products (ASSPs) targeted at selected market segments. For example, the RISC core can be integrated with peripherals such as an LCD controller, multimedia controllers, and an

external memory interface to empower OEMs to develop smaller, more cost-effective handheld devices with long battery life, with the performance to run rich multimedia applications. Or the microarchitecture could be surrounded by high-bandwidth PCI interfaces, memory controllers, and networking micro-engines to provide a highly integrated, low-power, I/O, or network processor.

1.2.2 Intel XScale® Microarchitecture Features

- Superpipelined RISC technology achieves high speed and low power
- Wireless Intel Speedstep® technology allows on-the-fly voltage and frequency scaling to enable applications to use the right blend of performance and power
- Media processing technology enables the MAC coprocessor perform two simultaneous 16-bit SIMD multiplies with 64-bit accumulation for efficient media processing
- Power management unit provides power savings via multiple low-power modes
- 128-entry branch target buffer keeps pipeline filled with statistically correct branch choices
- 32-Kbyte instruction cache (I-cache) keeps local copy of important instructions to enable high performance and low power
- 32-Kbyte data cache (D-cache) keeps local copy of important data to enable high performance and low power
- 2-Kbyte mini-data cache avoids “thrashing” of the D-cache for frequently changing data streams
- 32-entry instruction memory management unit enables logical-to-physical address translation, access permissions, I-cache attributes
- 32-entry data memory management unit enables logical-to-physical address translation, access permissions, D-cache attributes
- 4-entry fill and pend buffers promote core efficiency by allowing “hit-under-miss” operation with data caches
- Performance monitoring unit furnishes two 32-bit event counters and one 32-bit cycle counter for analysis of hit rates
- Debug unit uses hardware breakpoints and 256-entry trace-history buffer (for flow change messages) to debug programs
- 32-bit coprocessor interface provides high-performance interface between core and coprocessors
- 8-entry write buffer allows the core to continue execution while data is written to memory

See the *Intel XScale® Microarchitecture Users Guide* for additional information.

1.2.3 Intel® Wireless MMX™ technology

The Intel XScale® Microarchitecture has attached to it a coprocessor to accelerate multimedia applications. This coprocessor implements Intel® Wireless MMX™ technology, which is characterized by a 64-bit Single Instruction Multiple Data (SIMD) architecture and compatibility with the integer functionality of Intel® MMX™ technology and Streaming SIMD Extensions (SSE) instruction sets. The key features of this coprocessor are:

- 30 new media processing instructions
- 64-bit architecture with up to eight-way SIMD operations
- 16 x 64-bit register file

- SIMD PSR flags with group conditional execution support
- Instruction support for SIMD, SAD, and MAC
- Instruction support for alignment and video
- Intel® MMX™ technology and SSE integer compatibility
- Superset of existing Intel XScale® Microarchitecture media processing instructions

See the *The Complete Guide to Intel® Wireless MMX™ Technology* for more details.

1.2.4 Memory Architecture

1.2.4.1 Caches

There are two caches:

- Data cache – The PXA27x processor supports 32 Kbytes of data cache.
- Instruction Cache – The PXA27x processor supports 32 Kbytes of instruction cache.

1.2.4.2 Internal Memories

The key features of the PXA27x processor internal memory are:

- 256 Kbytes of on-chip SRAM arranged as four banks of 64 Kbytes
- Bank-by-bank power management with automatic power management for reduced power consumption
- Byte write support

1.2.4.3 External Memory Controller

The PXA27x processor supports a memory controller for external memory which can access:

- SDRAM up to 104 MHz at 1.8 Volts.
- Flash memories
- Synchronous ROM
- SRAM
- Variable latency input/output (VLIO) memory
- PC card and compact flash expansion memory

1.2.5 Processor Internal Communications

The PXA27x processor supports a hierarchical bus architecture. A system bus supports high-bandwidth peripherals, and a slower peripheral bus supports peripherals with lower data throughputs.

1.2.5.1 System Bus

- Interconnection between the major key components is through the internal system bus.

- 64-bit wide, address- and data-multiplexed bus.
- System bus allows split transactions, increasing the maximum data throughput in the system.
- Different burst sizes are allowed; up to 4 data phases per transactions (that is, 32 bytes). The burst size is set in silicon for each peripheral and is not configurable.
- The system bus can operate up to 208 MHz at different frequency ratios with respect to the Intel XScale® core. The frequency control of the system bus is pivotal to striking a balance between the preferred performance and power consumption.

1.2.5.2 Peripheral Bus

The peripheral bus is a single master bus. The bus master arbitrates between the Intel XScale® core and the DMA controller with a pre-defined priority scheme between them. The peripheral bus runs at 26 MHz and is used by the low-bandwidth peripherals.

1.2.5.3 Peripherals in the Processor

The PXA27x processor has a rich set of peripherals. The list of peripherals and key features are described in the subsections below.

1.2.5.3.1 LCD Display Controller

The LCD controller supports single- or dual-panel LCD displays. Color panels without internal frame buffers up to 262,144 colors (18 bits) are supported. Color panels with internal frame buffers up to 16,777,216 colors (24 bits) are supported. Monochrome panels up to 256 gray-scale levels (8 bits) are supported.

1.2.5.3.2 DMA Controller

The PXA27x processor has a high-performance DMA controller supporting memory-to-memory transfers, peripheral-to-memory, and memory-to-peripheral device transfers. It has support for 32 channels and up to 63 peripheral devices. The controller can perform descriptor chaining. The DMA controller supports descriptor-fetch, no-descriptor-fetch, and descriptor-chaining.

1.2.5.3.3 Other Peripherals

The PXA27x processor offers the following peripheral support:

- USB client controller with 23 programmable endpoints (compliant with USB Revision 1.1).
- USB host controller (USB Rev. 1.1 compatible), which supports both low-speed and full-speed USB devices through a built-in DMA controller.
- Intel® Quick Capture Interface, which provides a connection between the processor and a camera-image sensor.
- Infrared communication port (ICP) which supports 4 Mbps data rate compliant with Infrared Data Association (IrDA) standard.
- I^2C serial bus port, which is compliant with I^2C standard (also supports arbitration between multiple-masters).
- AC97 CODEC interface (compliant with AC97 2.0) supporting multiple independent channels (different channels are used for stereo PCM In, stereo PCM Out, MODEM Out, MODEM-In and mono mic-in).

- I^2S audio CODEC interface.
- Three flexible synchronous serial ports.
- Multimedia card controller that supports the MMC, SD, and SDIO protocols.
- Three UARTs compatible with 16550 and 16750 standard.
- Memory Stick host controller - compliant with Memory Stick V1.3 standard.
- USIM card interface (compliant with ISO standard 7816-3 and 3G TS 31.101)
- MSL – the physical interface of communication subsystems for mobile or wireless platforms. The operating system and application software uses this to communicate between each other.
- Keypad interface that supports both direct key as well as matrix key.
- Real-time clock (RTC) controller that provides a general-purpose, real-time reference clock for use by the system.
- Pulse-width modulator (PWM) controller, which generates four independent PWM outputs.
- Interrupt controller, which identifies and controls the interrupt sources available to the processor.
- OS timers controller, which provides a set of timer channels that allow software to generate timed interrupts or wake-up events.
- General-purpose I/O (GPIO) controller for use in generating and capturing application-specific input and output signals. Each of the 121[1] GPIOs may be programmed as an output, an input (or as bidirectional for certain alternate functions).

1.2.6 Wireless Intel Speedstep® technology

Wireless Intel Speedstep® technology advances the capabilities of Intel® Dynamic Voltage Management—a function already built into the Intel XScale® Microarchitecture—by incorporating three new low-power states: deep idle, standby, and deep sleep. The technology is able to change both voltage and frequency on-the-fly by switching the processor into the various low-power modes, saving additional power while still providing the necessary performance to run rich applications.

The PXA27x processor provides a rich set of flexible power-management controls for a wide range of usage models, while enabling very low-power operation. The key features include:

- Five reset sources:
  - — Power-on
  - — Hardware
  - — Watchdog
  - — GPIO
  - — Exit from sleep mode
- Three clock-speed controls to adjust frequency:
  - — Turbo mode
  - — Divisor mode

1. 121 GPIOs are available on the MCP package. The discrete package only has 119 GPIOs bonded out.

  - Fast bus mode
- Switchable clock source
- Functional clock gating
- Programmable frequency-change capability
- Six power modes to control power consumption:
  - Normal
  - Idle
  - Deep idle
  - Standby
  - Sleep
  - Deep sleep
- Programmable I^2C-based external regulator interface to support voltage changing.

See the *Intel® PXA27x Processor Family Developer's Manual* for more details.

1.3 Intel XScale® Microarchitecture Compatibility

The Intel XScale® Microarchitecture is ARM* Version 5 (V5TE) architecture compliant. The PXA27x processor implements the integer instruction set architecture of ARM*V5TE.

Backward compatibility for user-mode applications is maintained with the earlier generations of StrongARM* and Intel XScale® Microarchitecture processors. Operating systems may require modifications to match the specific Intel XScale® Microarchitecture hardware features and to take advantage of the performance enhancements added to this core.

Memory map and register locations are backward-compatible with the previous Intel XScale® Microarchitecture hand-held products.

The Intel® Wireless MMX™ technology instruction set is compatible with the standard ARM* coprocessor instruction format (See *The Complete Guide to Intel® Wireless MMX™ Technology* for more details).

1.3.1 PXA27x Processor Performance Features

The PXA27x processor includes the following performance features:

- 32-Kbyte instruction cache
- 32-Kbyte data cache
- Intel® Wireless MMX™ technology with sixteen 64-bit registers and optimized instructions for video and multi-media applications.
- Internal SRAM of 256 Kbytes, usable for both code and data
- Capability of locking entries in the instruction or data caches
- 2-Kbyte mini-data cache, separate from the data cache
- Use of virtual address indices (or tags) for the L1 caches and the mini-data cache

- Separate instruction and data translation lookaside buffers (TLBs), each with 32 entries
- Capability of locking entries in the TLBs
- 16-channel DMA engine with transfer-size control and descriptor chaining
- PID register for fast virtual address remapping
- Vector remap
- Interrupt controller that offers faster interrupt latency with the help of a programmable priority sorting mechanism.
- Extensions to the exception model to include imprecise data and instruction preload aborts
- Access control to other coprocessors
- Enhanced set of supported cache-control options
- Branch target buffer for dynamic-branch prediction
- Performance monitoring unit
- Software-debug support, including instruction and data breakpoints, a serial debug link via the JTAG interface and a 256-entry trace buffer
- Integrated memory controller with support for SDRAM, flash memory, synchronous ROM, SRAM, variable-latency I/O (VLIO) memory, PC card, and compact-flash expansion memory.

§§

Microarchitecture Overview 2

2.1 Introduction

This chapter contains an overview of Intel XScale® Microarchitecture and Intel® Wireless MMX™ Technology. The Intel XScale® Microarchitecture includes a superpipelined RISC architecture with an enhanced memory pipeline. The Intel XScale® Microarchitecture instruction set is based on ARM* V5TE architecture; however, the Intel XScale® Microarchitecture includes new instructions. Code developed for the Intel® StrongARM* SA-110 (SA-110), Intel® StrongARM* SA-1100 (SA-1100), and Intel® StrongARM* SA-1110 (SA-1110) microprocessors is portable to Intel XScale® Microarchitecture-based processors. However, to obtain the maximum performance, optimize the code for the Intel XScale® Microarchitecture using the techniques presented in this document.

2.2 Intel XScale® Microarchitecture Pipeline

This section provides a brief description of the structure and behavior of Intel XScale® Microarchitecture pipeline.

2.2.1 General Pipeline Characteristics

The following sections discuss general pipeline characteristics.

2.2.1.1 Pipeline Organization

The Intel XScale® Microarchitecture has a 7-stage pipeline operating at a higher frequency than its predecessors, allowing for greater overall performance. The Intel XScale® Microarchitecture single-issue superpipeline consists of a main execution pipeline, a multiply-accumulate {MAC} pipeline, and a memory access pipeline. Figure 2-1 shows the pipeline organization with the main execution pipeline shaded.

Figure 2-1. Intel XScale® Microarchitecture RISC Superpipeline

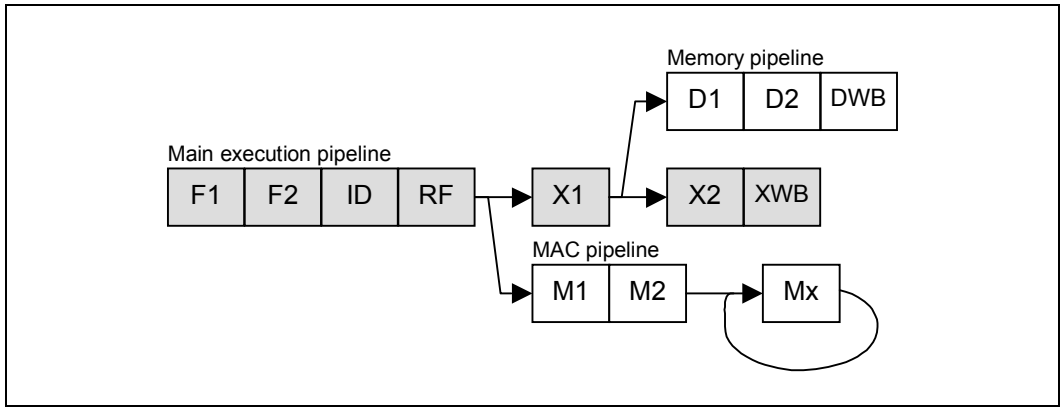


Table 2-1 gives a brief description of each pipe stage and a reference for additional information.

Table 2-1. Pipelines and Pipe Stages

Pipe / Pipestage	Description	For More Information
Main Execution Pipeline	Handles data processing instructions	Section 2.2.3
• IF1/IF2	Instruction Fetch	Section 2.2.3.1
• ID	Instruction Decode	Section 2.2.3.2
• RF	Register File / Operand Shifter	Section 2.2.3.3
• X1	ALU Execute	Section 2.2.3.4
• X2	State Execute	Section 2.2.3.5
• XWB	Write-back	Section 2.2.3.6
Memory Pipeline	Handles load/store instructions	Section 2.2.4
• D1/D2	Data cache access	Section 2.2.4.1
• DWB	Data cache writeback	Section 2.2.5.1
MAC Pipeline	Handles all multiply instructions	Section 2.2.5
• M1-M5	Multiplier stages	Section 2.2.5
• MWB (not shown)	MAC write-back occurs during M2-M5	Section 2.2.5

2.2.1.2 Out-of-Order Completion

While the pipeline is scalar and single issue, instructions occupy all three pipelines at once. The main execution pipeline, memory, and MAC pipelines have different execution times because they are not lock stepped. Sequential consistency of instruction execution relates to two aspects: first, the order instructions are completed, and second, the order memory is accessed due to load and store instructions. The Intel XScale® Microarchitecture only preserves a weak processor consistency because instructions complete out of order (assuming no data dependencies exist).

The Intel XScale® Microarchitecture can buffer up to four outstanding reads. If load operations miss the data cache, subsequent instructions complete independently. This operation is called a *hit-under-miss operation*.

2.2.1.3 Use of Bypassing

The pipeline makes extensive use of bypassing to minimize data hazards. To eliminate the need to stall the pipeline, bypassing allows results forwarding from multiple sources.

In certain situations, the pipeline must stall because of register dependencies between instructions. A register dependency occurs when a previous MAC or load instruction is about to modify a register value that has not returned to the register file. Core bypassing allows the current instruction to execute when the previous instruction's results are available without waiting for the register file to update.

2.2.2 Instruction Flow Through the Pipeline

With the exception of the MAC unit, the pipeline issues one instruction per clock cycle. Instruction execution begins at the F1 pipestage and completes at the WB pipestage.

Although a single instruction is issued per clock cycle, all three pipelines are processing instructions simultaneously. If there are no data hazards, each instruction complete independently of the others.

2.2.2.1 ARM* V5TE Instruction Execution

Figure 2-1 uses arrows to show the possible flow of instructions in the pipeline. Instruction execution flows from the F1 pipestage to the RF pipestage. The RF pipestage issues a single instruction to either the X1 pipestage or the MAC unit (multiply instructions go to the MAC, while all others continue to X1). This means that M1 or X1 are idle.

After calculating the effective addresses in XI, all load and store instructions route to the memory pipeline.

The ARM* V5TE branch and exchange (BX) instruction (used to branch between ARM* and THUMB* code) flushes the entire pipeline. If the processor is in THUMB* mode, the ID pipestage dynamically expands each THUMB* instruction into a normal ARM* V5TE RISC instruction and normal execution resumes.

2.2.2.2 Pipeline Stalls

Pipeline stalls can seriously degrade performance. The primary reasons for stalls are register dependencies, load dependencies, multiple-cycle instruction latency, and unpredictable branches. To help maximize performance, it is important to understand some of the ways to avoid pipeline stalls. The following sections provide more detail on the nature of the pipeline and ways of preventing stalls.

2.2.3 Main Execution Pipeline

2.2.3.1 F1 / F2 (Instruction Fetch) Pipestages

The job of the instruction fetch stages F1 and F2 is to present the next instruction to be executed to the ID stage. Two important functional units residing within the F1 and F2 stages are the BTB and IFU.

- Branch Target Buffer (BTB)

  The BTB provides a 128-entry dynamic branch-prediction buffer. An entry in the BTB is created when a B or BL instruction branch is taken for the first time. On sequential executions of the branch instruction at the same address, the next instruction loaded into the pipeline is predicted by the BTB. Once the branch-type instruction reaches the X1 pipestage, its target address is known. Execution continues without stalling if the target address is the same as the BTB predicted address. If the address is different from the address that the BTB predicted, the pipeline is flushed, execution starts at the new target address, and the branch history is updated in the BTB.

- Instruction Fetch Unit (IFU)

  The IFU is responsible for delivering instructions to the instruction decode (ID) pipestage. It delivers one instruction word each cycle (if possible) to the ID. The instruction could come from one of two sources: instruction cache or fetch buffers.

2.2.3.2 Instruction Decode (ID) Pipestage

The ID pipestage accepts an instruction word from the IFU and sends register decode information to the RF pipestage. The ID can accept a new instruction word from the IFU on every clock cycle in which there is no stall. The ID pipestage is responsible for:

- General instruction decoding (extracting the opcode, operand addresses, destination addresses and the offset).
- Detecting undefined instructions and generating an exception.
- Dynamic expansion of complex instructions into sequence of simple instructions. Complex instructions are defined as ones that take more than one clock cycle to issue, such as LDM, STM, and SWP.

2.2.3.3 Register File / Shifter (RF) Pipestage

The main function of the RF pipestage is to read and write to the register file unit (RFU). It provides source data to the following:

- X1 for ALU operations
- MAC for multiply operations
- Data cache for memory writes
- Coprocessor interface

The ID unit decodes the instruction and specifies the registers accessed in the RFU. Based on this information, the RFU determines if it needs to stall the pipeline due to a register dependency. A register dependency occurs when a previous instruction is about to modify a register value that has not been returned to the RFU and the current instruction needs to access that same register. If no dependencies exist, the RFU selects the appropriate data from the register file and passes it to the next pipestage. When a register dependency does exist, the RFU keeps track of the unavailable register. The RFU stops stalling the pipe when the result is returned.

The ARM* architecture specifies one of the operands for data processing instructions as the shifter operand. A 32-bit shift can be performed on a value before it is used as an input to the ALU. This shifter is located in the second half of the RF pipestage.

2.2.3.4 Execute (X1) Pipestages

The X1 pipestage performs these functions:

- ALU calculations – the ALU performs arithmetic and logic operations, as required for data processing instructions and load/store index calculations.
- Determine conditional instruction executions – the instruction condition is compared to the CPSR prior to execution of each instruction. Any instruction with a false condition is cancelled and does not cause any architectural state changes, including modifications of registers, memory, and PSR.
- Branch target determinations – the X1 pipestage flushes all instructions in the previous pipestages and sends the branch target address to the BTB if a branch is mispredicted by the BTB. The flushing of these instructions restarts the pipeline.

2.2.3.5 Execute 2 (X2) Pipestage

The X2 pipestage contains the program status registers (PSR). This pipestage selects the data to be written to the RFU in the WB cycle including the following items.

The X2 pipestage contains the current program status register (CPSR). This pipestage selects what is written to the RFU in the WB cycle including program status registers.

2.2.3.6 Write-Back (WB)

When an instruction reaches the write-back stage, it is considered complete. Instruction results are written to the RFU.

2.2.4 Memory Pipeline

The memory pipeline consists of two stages, D1, and D2. The data cache unit (DCU) consists of the data-cache array, mini-data cache, fill buffers, and write buffers. The memory pipeline handles load and store instructions.

2.2.4.1 D1 and D2 Pipestage

Operation begins in D1 after the X1 pipestage calculates the effective address for loads and stores. The data cache and mini-data cache return the destination data in the D2 pipestage. Before data is returned in the D2 pipestage, sign extension and byte alignment occurs for byte and half-word loads.

2.2.4.1.1 Write Buffer Behavior

The Intel XScale® Microarchitecture has enhanced write performance by the use of write coalescing. Coalescing is combining a new store operation with an existing store operation already resident in the write buffer. The new store is placed in the same write buffer entry as an existing store when the address of new store falls in the 4-word aligned address of the existing entry.

The core can coalesce any of the four entries in the write buffer. The Intel XScale® Microarchitecture has a global coalesce disable bit located in the Control register (CP15, register 1, opcode_2=1).

2.2.4.1.2 Read Buffer Behavior

The Intel XScale® Microarchitecture has four fill buffers that allow four outstanding loads to the cache and external memory. Four outstanding loads increases the memory throughput and the bus efficiency. This feature can also be used to hide latency. Page table attributes affect the load behavior; for a section with C=0, B=0 there is only one outstanding load from the memory. Thus, the load performance for a memory page with C=0, B=1 is significantly better compared to a memory page with C=0, B=0.

2.2.5 Multiply/Multiply Accumulate (MAC) Pipeline

The multiply-accumulate (MAC) unit executes the multiply and multiply-accumulate instructions supported by the Intel XScale® Microarchitecture. The MAC implements the 40-bit Intel XScale® Microarchitecture accumulator register acc0 and handles the instructions that transfers its value to and from general-purpose ARM* registers.

The MAC has several important characteristics:

- The MAC is not a true pipeline. The processing of a single instruction requires use of the same data-path resources for several cycles before a new instruction is accepted. The type of instruction and source arguments determine the number of required cycles.
- No more than two instructions can concurrently occupy the MAC pipeline.
- When the MAC is processing an instruction, another instruction cannot enter M1 unless the original instruction completes in the next cycle.
- The MAC unit can operate on 16-bit packed signed data, which reduces register pressure and memory traffic size. Two 16-bit data items can be loaded into a register with one LDR.
- The MAC can achieve throughput of one multiply per cycle when performing a 16-by-32-bit multiply.
- ACC registers in the Intel XScale® Microarchitecture can be up to 64 bits in future implementations. Code should be written to depend on the 40-bit nature of the current implementation.

2.2.5.1 Behavioral Description

The execution of the MAC unit starts at the beginning of the M1 pipestage. At this point, the MAC unit receives two 32-bit source operands. Results are completed N cycles later (where N is dependent on the operand size) and returned to the register file. For more information on MAC instruction latencies, refer to Section 4.8, "Instruction Latencies for Intel XScale® Microarchitecture".

An instruction occupying the M1 or M2 pipestages occupies the X1 and X2 pipestage, respectively. Each cycle, a MAC operation progresses for M1 to M5. A MAC operation may complete anywhere from M2-M5.

2.2.5.2 Perils of Superpipelining

The longer pipeline has several consequences worth considering:

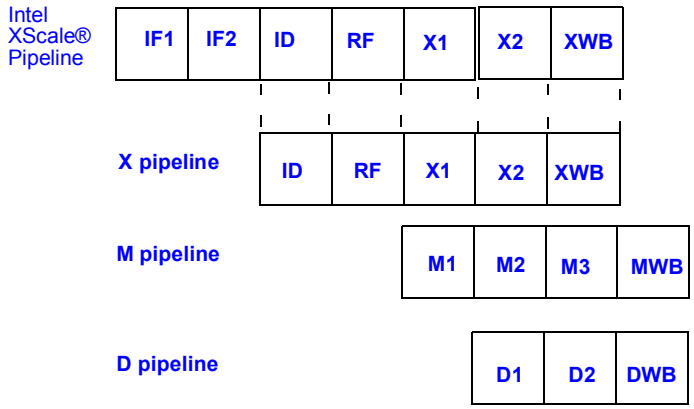
- Larger branch misprediction penalty (four cycles in the Intel XScale® Microarchitecture instead of one in StrongARM* Architecture).
- Larger load use delay (LUD) — LUDs arise from load-use dependencies. A load-use dependency gives rise to a LUD if the result of the load instruction cannot be made available by the pipeline in time for the subsequent instruction. To avoid these penalties, an optimizing compiler should take advantage of the core's multiple outstanding load capability (also called hit-under-miss) as well as finding independent instructions to fill the slot following the load.
- Certain instructions incur a few extra cycles of delay with the Intel XScale® Microarchitecture as compared to StrongARM* processors (LDM, STM).
- Decode and register file lookups are spread out over two cycles with the Intel XScale® Microarchitecture, instead of one cycle in predecessors.

2.3 Intel® Wireless MMX™ Technology Pipeline

As the Intel® Wireless MMX™ Technology is tightly coupled with the Intel XScale® Microarchitecture; the Intel® Wireless MMX™ Technology pipeline follows the similar pipeline structure as the Intel XScale® Microarchitecture. Figure 2-2 shows the Intel® Wireless MMX™ Technology pipeline, which contains three independent pipeline threads:

- X pipeline - Execution pipe
- M pipeline - Multiply pipe
- D pipeline - Memory pipe

Figure 2-2. Intel® Wireless MMX™ Technology Pipeline Threads and relation with Intel XScale® Microarchitecture Pipeline



2.3.1 Execute Pipeline Thread

2.3.1.1 ID Stage

The ID pipe stage is where decoding of Intel® Wireless MMX™ Technology instructions commences. Because of the significance of the transit time from Intel XScale® Microarchitecture in the ID pipe stage, only group decoding is performed in the ID stage, with the remainder of the decoding being completed in the RF stage. However, it is worth noting that the register address decoding is fully completed in the ID stage because the register file needs to be accessed at the beginning of the RF stage.

All instructions are issued in a single cycle, and they pass through the ID stage in one cycle if no pipeline stall occurs.

2.3.1.2 RF Stage

The RF stage controls the reading/writing of the register file, and determines if the pipeline has to stall due to data or resource hazards. Instruction decoding also continues at the RF stage and completes at the end of the RF stage. The register file is accessed for reads in the high phase of the clock and accessed for writes in the low phase.If data or resource hazards are detected, the Intel®

Wireless MMX™ Technology stalls Intel XScale® Microarchitecture. Note that control hazards are detected in the Intel XScale® Microarchitecture, and a flush signal is sent from the core to the Intel® Wireless MMX™ Technology.

2.3.1.3 X1 Stage

The X1 stage is also known as the *execution stage,* which is where most instructions begin being executed. All instructions are conditionally executed and that determination occurs at the X1 stage in the Intel XScale® Microarchitecture. A signal from the core is required to indicate whether the instruction being executed is committed. In other words, an instruction being executed at the X1 stage may be canceled by a signal from the core. This signal is available to the Intel® Wireless MMX™ Technology in the middle of the X1 pipe stage.

2.3.1.4 X2 Stage

The Intel® Wireless MMX™ Technology supports saturated arithmetic operations. Saturation detection is completed in the X2 pipe stage.

If the Intel XScale® Microarchitecture detects exceptions and flushes in the X2 pipe stage, Intel® Wireless MMX™ Technology also flushes all the pipeline stages.

2.3.1.5 XWB Stage

The XWB stage is the last stage of the X pipeline, where a final result calculated in the X pipeline is written back to the register file.

2.3.2 Multiply Pipeline Thread

2.3.2.1 M1 Stage

The M pipeline is separated from the X pipeline. The execution of multiply instructions starts at the beginning of the M1 stage, which aligns with the X1 stage of the X pipeline. While the issue cycle for multiply operations is one clock cycle, the result latency is at least three cycles. Certain instructions such as TMIA, WMAC, WMUL, WMADD spend two M1 cycles since the Intel® Wireless MMX™ Technology has only two 16x16 multiplier arrays. Booth encoding and first-level compression occur in the M1 pipe stage.

2.3.2.2 M2 Stage

Additional compression occurs in the M2 pipe stage, and the lower 32 bits of the result are calculated with a 32 bit adder.

2.3.2.3 M3 Stage

The upper 32 bits of the result are calculated with a 32-bit adder.

2.3.2.4 MWB Stage

The MWB stage is the last stage of the M pipeline, which is where a final result calculated in the M pipeline is written back to the register file.

A forwarding path from the MWB stage to the RF stage serves as a non-critical bypass. Critical and reasonable logic insertion are allowed.

2.3.3 Memory Pipeline Thread

2.3.3.1 D1 Stage

In the D1 pipe stage, the Intel XScale® Microarchitecture provides a virtual address that is used to access the data cache. There is no logic inside the Intel® Wireless MMX™ Technology in the D1 pipe stage.

2.3.3.2 D2 Stage

The D2 stage is where load data is returned. Load data comes from either data cache or external memory, with external memory having the highest priority. The Intel® Wireless MMX™ Technology needs to bridge incoming 32-bit data to internal 64-bit data.

2.3.3.3 DWB Stage

The DWB stage—the last stage of the D pipeline—is where load data is written back to the register file.

§§

§§

System Level Optimization 3

This chapter describes relevant performance considerations that developers and system designers need to be aware of to use the Intel® PXA27x Processor Family (PXA27x processor) efficiently.

3.1 Optimizing Frequency Selection

The PXA27x processor offers a range of combinations of core, system bus and memory clock speed. Choose a high core frequency for operations that are CPU or core bound. These operations include computations and operations on data in the cache. Choose a high system-bus, memory clock, and SDRAM clock for operations that are memory bound. These operations include *memcpy()* and other functions that depend on memory accesses. For example, the frequency point of core:system:memory:SDRAM clock at 312:208:208:104 offers better memory performance than 312:104:104:104, even though core performance should be nearly identical.

The range of possible frequency selections is listed in the clocks and power manager section of the *Intel® PXA27x Processor Family Developer's Manual*. Choose the frequency combination to fit the target application mix as well as use the frequency points that will be supported during the silicon manufacturing process. Consult a local Intel field applications engineer (FAE) for details on these product points.

3.2 Memory System Optimization

3.2.1 Optimal Setting for Memory Latency and Bandwidth

Because the PXA27x processor has a multi-transactional internal bus, there are latencies involved with accesses to and from the Intel XScale® core. The internal bus, also called the system bus, allows many internal operations to occur concurrently such as LCD, DMA controller and related data transfers. Table 3-1 and Table 3-2 list latencies and throughputs associated with different frequencies. The throughput reported in the tables is only measuring load throughput.

Table 3-1. External SDRAM Access Latency and Throughput for Different Frequencies (Silicon Measurement Pending)

Core Clock Speed (MHz) (up to)	Run Mode Frequency (MHz) (up to)	System Bus Clock Speed (MHz) (up to)	Memory Clock Speed (MHz) (up to)	Memory Latency (core cycles)	Load Throughput from Memory (MBytes/ Sec)
104	104	104	104	17	205
208	208	208	104	21	326
312	208	208	104	30	343

Table 3-2. Internal SRAM Access Latency and Throughput for Different Frequencies (Silicon Measurement Pending)

Core Clock Speed (MHz) (up to)	Run Mode Frequency (MHz) (up to)	System Bus Clock Speed (MHz) (up to)	Memory Clock Speed (MHz) (up to)	Memory Latency (core cycles)	Load Throughput from Memory (MBytes/ Sec)†
104	104	104	104	14	236
208	208	208	104	14	472
312	208	208	104	21	473

† Store throughput is similar

- Setting wait-states for static memory

  For static memory, it is important to use the correct number of wait states to achieve optimal performance. The *Intel® PXA27x Processor Family Developer's Manual* explains the possible values in the MSCx registers. The bitfields RBUFx, RRRx, RDNx, and RDFx in these registers control wait states and set up the access mode used. For flash memory that supports asynchronous page reads, these modes provide improvements in reading and executing from flash.

- Enabling burst reads for synchronous flash memory

  Modern synchronous flash memory supports asynchronous page-mode reads and faster synchronous burst-mode reads. Page-mode and burst-mode reads help improve the bus bandwidth of flash transfers on the external bus shared with SDRAM, which also improves the performance of flash file systems. Typically, asynchronous mode is enabled at power-on reset, both in the processor and in the flash memory. To enable synchronous burst mode, follow these steps:

  1. Set the PXA27x processor's Synchronous Static Memory Configuration register (SXCNFG) to enable burst-of-8 or burst-of-16 mode. See the *Intel® PXA27x Processor Family Developer's Manual* for details.

  2. After enabling the processor, similarly enable the flash memory for burst reads through its Read Configuration register (RCR), which is described in the Intel StrataFlash® Wireless Memory datasheet.

     Be careful when changing to synchronous mode. The instructions for the RCR configuration sequence must either be in RAM or guaranteed not to fetch from the flash during the RCR programming operation. Accesses to the flash memory cannot occur until both the PXA27x processor and the flash memory are in synchronous read mode.

  3. Mark the address space of the flash device as cacheable to allow the PXA27x processor memory controller to fill cache lines from flash in burst reads, enabling faster performance during reads from flash.

  4. To maintain cache coherency during writes to flash, use a layer of software to monitor write/erase operations to flash, and to invalidate the cache lines affected by those operations.

  These steps are typically enabled in board-support packages (BSPs) and flash file-system software. Check with your FAE on their availability.

- SDRAM refresh

  The DRI field in the MDREFR register controls the interval between SDRAM auto-refresh cycles. Set the interval to meet the minimum requirements specified by the SDRAM

manufacturer. Avoid excessive refresh cycles, which reduce the bandwidth available for accessing the SDRAM.

- SDRAM CAS latency

  For SDRAM, the key parameter is the CAS latency. Lower CAS latency gives higher performance. Most current SDRAM supports a CAS latency of two.

- Setting of the APD bit

  Use of the APD bit in the memory controller can save power; however, it can also increase the memory latency. Clear the APD bit for high performance.

- Buffer Strength registers

  The output drivers for the PXA27x processor external memory bus have programmable strength settings. This feature allows for simple, software-based control of the output driver impedance for the external memory bus. Use these registers to match the driver strength of the PXA27x processor to external memory bus. Set the buffer strength to the lowest possible setting (minimum drive strength) that still allows for reliable memory system performance. This also minimizes the power use of the external memory bus, which is a major component of total system power. Refer to the Programmable Output Buffer Strength registers described in the *Intel® PXA27x Processor Family Developer's Manual*, for more information.

3.2.2 System Bus and Alternate Memory Clock Settings

The latency and throughput of memory transactions in the PXA27x processor is governed largely by the speeds of the internal system bus and memory controller, and external SDRAM. The CLKCFG[B] field controls the speed of the system bus, and the CCCR[A] field controls the speed of the memory bus.

CLKCFG[B] controls whether fast-bus mode is enabled. When B is set, the system-bus frequency is equal to the run-mode frequency indicated in CCCR. When B is clear, the system-bus frequency is equal to half the run-mode frequency indicated in the CCCR. Set the B bit to enable the fastest system-bus frequency.

The CCCR[A] field enables an alternate set of memory-controller clock selections. See the "CCCR Bit Definitions" table in the *Intel® PXA27x Processor Family Developer's Manual*. When this bit is set the memory clock speed is expanded to allow it to be set as high as 208 MHz. When cleared the maximum memory clock speed is 130 MHz. The memory clock refers to the speed of the memory controller, which governs access between the internal system bus and external memory. Set the A bit to enable the fastest memory-controller frequency.

If CCCR[A] is cleared, use the "Core PLL Output Frequencies for 13-MHz Crystal with CCCR[A] = 0" table in the *Intel® PXA27x Processor Family Developer's Manual* when making the clock setting selections. If CCCR[A] is set, use the "Core PLL Output Frequencies for 13-MHz Crystal With B=0 and CCCR[A] = 1" table and the "Core PLL Output Frequencies for 13-MHz Crystal With B=1 and CCCR[A] = 1" table in the *Intel® PXA27x Processor Family Developer's Manual* instead.

3.2.3 Page Table Configuration

Three bits for each page are used to configure each memory page cache behavior. Different values of X,C,B determine the caching, reading and writing, and buffering policies of the pages.

3.2.3.1 Page Attributes For Instructions

When examining these bits in a descriptor, the instruction cache uses only the C bit. If the C bit is clear, the instruction cache considers a code fetch from that memory to be noncacheable, and does not fill a cache entry. If the C bit is set, then fetches from the associated memory region are cached.

3.2.3.2 Page Attributes For Data Access

For data access, all three attributes are important. If the X bit for a descriptor is zero, the C and B bits operate as defined by the ARM* architecture. This behavior is detailed in Table 3-3.

If the X bit for a descriptor is one, the C and B bits behave differently, as shown in Table 3-4. The load and store buffer behavior in Intel XScale® Microarchitecture is explained in Section 2.2.4.1.1, "Write Buffer Behavior" and Section 2.2.4.1.2, "Read Buffer Behavior"

Table 3-3. Data Cache and Buffer Behavior when X = 0

C B	Cacheable?	Load Buffering and Write Coalescing?	Write Policy	Line Allocation Policy	Notes
0 0	N	N	—	—	Stall until complete†
0 1	N	Y	—	—	
1 0	Y	Y	Write-through	Read Allocate	
1 1	Y	Y	Write-back	Read Allocate	

† Normally, the processor continues executing after a data access if no dependency on that access is encountered. With this setting, the processor stalls execution until the data access completes. This guarantees to software that the data access has taken effect by the time execution of the data access instruction completes. External data aborts from such accesses are imprecise.

Table 3-4. Data Cache and Buffer Behavior when X = 1

C B	Cacheable?	Load Buffering and Write Coalescing?	Write Policy	Line Allocation Policy	Notes
0 0	—	—	—	—	Unpredictable -- do not use
0 1	N	Y	—	—	Writes do not coalesce into buffers†
1 0	(Mini-data cache)	—	—	—	Cache policy is determined by MD field of Auxiliary Control register††
1 1	Y	Y	Write-back	Read/Write Allocate	

† Normally, "bufferable" writes can coalesce with previously buffered data in the same address range

†† Refer to *Intel XScale® Core Developer's Manual* and the *Intel® PXA27x Processor Family Developer's Manual* for a description of this register.

Note: The Intel XScale® Microarchitecture page attributes are different from the Intel® StrongARM* SA-1110 Microprocessor (SA-1110). The SA-1110 code may behave differently on PXA27x processor systems due to page-attribute differences. Table 3-5 describes the differences in the encoding of the C and B bits for data accesses. The main difference occurs when cacheable and nonbufferable data is specified (C=1, B=0); the SA-1110 uses this encoding for the mini-data cache while the Intel XScale® Microarchitecture uses this encoding to specify writethrough caching.

Another difference is when C=0, B=1, where the Intel XScale® Microarchitecture coalesces stores in the write buffer; the SA-1110 does not.

Table 3-5. Data Cache and Buffer operation comparison for Intel® SA-1110 and Intel XScale® Microarchitecture, X=0

Encoding	SA-1110 Function	Intel XScale® Microarchitecture Function
C=1,B=1	Cacheable in data cache; store misses can coalesce in write buffer	Cacheable in data cache, store misses can coalesce in write buffer
C=1,B=0	Cacheable in mini-data cache; store misses can coalesce in write buffer	Cacheable in data cache, with a write-through policy. Store misses can coalesce in write buffer
C=0,B=1	Noncacheable; no coalescing in write buffer, but can wait in write buffer	Noncacheable; stores can coalesce in the write buffer
C=0,B=0	Noncacheable; no coalescing in the write buffer, SA-110 stalls until this transaction is done	Noncacheable, no coalescing in the write buffer, Intel XScale® Microarchitecture stalls until the operation completes.

3.3 Optimizing for Instruction and Data Caches

Cache locking allows frequently used code to be locked in the cache. Up to 28 cache lines can be locked in a set, while the remaining four entries still participate in the round-robin replacement policy.

3.3.1 Increasing Instruction Cache Performance

The performance of the PXA27x processor is depends highly on the cache miss rate. Due to the complexity of the processor fetching instructions from external memory can have a large latency. Moreover, this cycle penalty becomes significant when the Intel XScale® core is running much faster than external memory. Executing non-cached instructions severely curtails processor performance so it is important to to minimize cache misses.

3.3.1.1 Round-Robin Replacement Cache Policy

Both the data and the instruction caches use a round-robin replacement policy to evict a cache line. The simple consequence of this is that every line will eventually be evicted, assuming a non-trivial program. The less obvious consequence is that predicting when and over which cache lines evictions occur is difficult. This information must be gained by experimentation using performance profiling.

3.3.1.2 Code Placement to Reduce Cache Misses

Code placement can greatly affect cache misses. One way to view the cache is to think of it as 32 sets, with each set spanning an address range of 1 kB. Each set is mapped as 32 ways, each way including 32 bytes, which span an address range of 1024 bytes. When running, the code maps into 32 blocks modulo 1024 of cache space. Any overused sets thrash the cache. The ideal situation is for the software tools to distribute the code on a temporal evenness over this space.

A compiler cannot perform this code distribution automatically. Most of the input needed to best estimate how to distribute the code comes from profiling followed by compiler-based two-pass optimizations.

3.3.1.3 Locking Code into the Instruction Cache

One important instruction-cache feature is the ability to lock code. Once locked into the instruction cache, the code is always available for fast execution. Another reason for locking critical code into cache is that with the round-robin replacement policy, eventually the code is evicted—even if it is a frequently executed function. Key code components to consider locking are:

- Interrupt handlers
- OS timer clock handlers
- OS critical code
- Time-critical application code

The disadvantage to locking code into the cache is that it reduces the cache size for the remainder of the program. How much code to lock is application dependent and requires experimentation to optimize.

Code placed into the instruction cache should be aligned on a 1024-byte boundary and placed sequentially together as tightly as possible so as not to waste memory space. Making the code sequential also ensures even distribution across all cache ways. Though it is possible to choose randomly located functions for cache locking, this approach runs the risk of locking multiple cache ways in one set and few or none in another set. This distribution unevenness can lead to excessive thrashing of instruction cache.

3.3.2 Increasing Data Cache Performance

There are different techniques which can be used to increase the data cache performance. These include, optimizing cache configuration and programming techniques etc. This section offers a set of system-level optimization opportunities; however program-level optimization techniques are equally important.

3.3.2.1 Cache Configuration

The Intel XScale® Microarchitecture allows users to define memory regions where cache policies are user defined. To support these various memory regions, OS configures the page-tables accordingly.

The performance of application code depends on which cache policy is used for data objects. A description of when to use a particular policy is described below.

If the application is running under an OS, then the OS may restrict the application from using certain cache policies.

3.3.2.1.1 Cache Configuration: Write-through and Write-back Cached Memory Regions

Write-back mode avoids some memory transactions by allowing data to collect in the data cache before eventually being written to memory when the cache line is evicted. When cache lines are evicted, the writes coalesce and are efficiently written to memory. This differs from writethrough mode where writes are always written to memory immediately. Writethrough memory regions

generate more data traffic on the bus and consume more power due to increased bus activity. The writeback policy is recommended whenever possible. However, in a multibus master environment, it may be necessary to use a writethrough policy if data is shared across multiple masters. In such a situation, all shared memory regions use a writethrough policy for all shared memory regions. Memory regions that are private to a particular master should use the writeback policy.

3.3.2.1.2 Cache Configuration: Read Allocate and Read-write Allocate Memory Regions

Writeback with read/write-allocate caches cause an additional read from the memory during a write miss. Subsequent read and write performance may be improved by more frequent cache hits. Most of the regular data and the stack for applications should be allocated to a read-write allocate region. Data that is write only (or data that is written to and subsequently not used for a long time) should be placed in a read-allocate region. Under the read allocate policy, if a cache write miss occurs, a new cache line is not allocated, and therefore does not evict data from the data cache. Memory-intensive operations like a memcopy can actually be slowed down by the extra reads required for the write-allocate policy.

3.3.2.1.3 Cache Configuration: Noncacheable Regions

Noncachable memory regions (X=0, C=0, B=0) are frequently needed for I/O devices. For these devices, the relevant device registers and memory spaces are mapped as noncacheable. In some cases, making the noncacheable regions bufferable (X=0, C=0, and B =1) can accelerate the memory performance due to write coalescing. There are cases where noncached memory regions must be set as nonbufferable (B=0):

- Any device where consecutive writes to the same address could be overwritten in the write buffer before reaching the target device (for example, FIFOs).
- Devices where read/write order to the device is required. When coalescing occurs, writes occur in numerical address order, not in the temporal order.

3.3.2.2 Creating Scratch RAM in the Internal SRAM

A very simple method for creating a fast-scratch RAM is to allocate a portion of the internal SRAM for this purpose to allow data mapped to this area to be accessed much more quickly than if it resided in external memory. Additionally, there are no considerations for cache locking, as are discussed in the next section, Section 3.3.2.3, “Creating Scratch RAM in Data Cache”.

This is the preferred method for creating scratch RAM for the PXA27x processor. It is generally preferable to keep as much of the data cache as possible available for its designated use: cache space. While access to the internal SRAM is slower than accessing data in cache, data in the scratch RAM generally do not suffer from the increased latency.

3.3.2.3 Creating Scratch RAM in Data Cache

Like the instruction cache, lines of the data cache can be locked. This can be thought of as converting parts of the cache into fast on-chip RAM. Access to objects in this on-chip RAM does not incur cache-miss penalties, thereby reducing the number of processor stalls. Application performance can be improved by locking data-cache lines and allocating frequently allocated variables to this space. Due to the Intel XScale® Microarchitecture round-robin replacement policy, all non-locked cache data is eventually evicted. Therefore, to prevent critical or frequently used data from being evicted, it can be allocated to on-chip RAM.

These variables are good candidates for allocating to the on-chip RAM:

- Frequently used global data used for storing context for context switching.
- Global variables that are accessed in time-critical functions such as interrupt-service routines.

When locking a memory region into the data cache to create on-chip RAM, ensure that all sets in the on-chip RAM area of the data cache have approximately the same number of ways locked. If some sets have more ways locked than others, the level of thrashing increases in some sets and leave other sets underutilized.

For example, consider three arrays: arr1, arr2, and arr3 of size 64 bytes each that are allocated to the on-chip RAM and assume that the address of arr1 is 0, address of arr2 is 1024, and the address of arr3 is 2048. All three arrays are within the same sets, set0 and set1. As a result, three ways in both sets set0 and set1 are locked, leaving 29 ways for use by other variables.

This issue can be overcome by allocating on-chip RAM data in sequential order. In the above example, allocating arr2 to address 64 and arr3 to address 128 allows the three arrays to use only one way in sets 0 through 5.

To reduce cache pollution between two processes and avoid frequent cache flushing during context switch, the OS could potentially lock critical data sections in the cache. The OS can also potentially offer the locking mechanism as a system function to its applications.

3.3.2.4 Reducing Memory Page Thrashing

Memory page thrashing occurs because of the nature of SDRAM. SDRAMs are typically divided into 4 banks. Each bank can have one selected page where a page address size for current memory components is often defined as 4 kB. Memory lookup time or latency time for a selected page address is currently 2 to 3 bus clocks. Thrashing occurs when subsequent memory accesses within the same memory bank access different pages. The memory page change adds 3 to 4 bus clock cycles to memory latency. This added delay extends the preload distance[1] correspondingly, making it more difficult to hide memory-access latencies. This type of thrashing can be resolved by placing the conflicting data structures into different memory banks or by paralleling the data structures such that the data resides within the same memory page. Ensure that instruction and data sections and the LCD frame buffer are in different memory banks or they will continually trash the memory page selection.

3.3.2.5 Using the Mini-Data Cache

The mini-data cache (X=1, C=1, B=0) is best used for data structures that have short temporal lives and/or cover vast amounts of data space. Addressing these types of data spaces from the data cache would corrupt much, if not all, of the data cache by evicting valuable data, which reduces performance. Placing this data instead in a mini-data cache memory region would help prevent data cache corruption while providing the benefits of cached accesses.

These examples of data could be assigned to mini-data cache:

- The stack space of a frequently occurring interrupt: The stack is used during the short duration of the interrupt only.
- Streaming media data: In many cases, the media steam data has limited time-span usage and would otherwise repeatedly evict the main data cache.

Overuse of the mini-data cache leads to thrashing the cache. This is easy to do because the mini-data cache has two ways per set. For example, see the following code snippet:

1. *Preload distance* is defined as the number of instructions required to preload data to avoid a core stall.

```
for (i=0; i<IMAX; i++)
{
    A[i] = B[i] + C[i];
}
```

If A, B, and C reside in a mini-data cache memory region, and each is array is aligned on a 1-kB boundary, the loop above quickly thrashes the cache.

The mini-data cache could also be used to keep frequently used tables cached. The advantage of keeping these in the minicache is two-fold. First, the data thrashing in the main cache does not thrash the frequently used tables and coefficients. Second, it saves main cache space from locking the critical blocks. For applications like MPEG-4, MP3, and GSM-AMR that handle big data streams, locking main data cache for these tables is not an efficient use of cache. During execution of such applications, these are some examples of tables which can effectively make use of the mini-data cache:

- Huffman tables
- Sine/cosine look-up tables
- Color-conversion look-up tables
- Motion compensation vector tables

3.3.2.6 Reducing Cache Conflicts, Pollution and Pressure

Cache pollution occurs when unused data is loaded in the cache and *cache pressure* occurs when data that is not temporal to the current process is loaded into the cache. Avoid excessive pre-loading and data locking. For an example, see Section 5.1.1.1.2, “Preload Loop Scheduling” on page 5-2. Increasing data locality through the use of programming techniques helps avoid these problems as well.

3.3.3 Optimizing TLB (Translation Lookaside Buffer) Usage

The Intel XScale® Microarchitecture offers hardware translation of virtual addresses to physical addresses, and 32 entries for instruction and data TLBs. The TLBs are used to cache translated physical addresses. Hardware page-table walking helps reduce the task of page translation for the OS, which may include hardware and software management of the TLBs. From a performance point of view, hardware TLBs are more efficient than software-managed TLBs. Intel recommends using hardware TLBs for page-table walking; however, to reduce data aborts, set the page-table attributes correctly.

During some usage scenarios, such as application launch and intensive frame-buffer access, TLB misses may cause a significant amount of overhead, resulting in reduced performance. If the target region of physical memory accessed is contiguous, one optimization technique is to map the pages of virtual memory in the largest size possible. For example, consider mapping the LCD frame buffer as 1-MB sections instead of 4-kB pages to reduce TLB-miss penalties, which helps increase graphics performance.

The Intel XScale® Microarchitecture allows individual entries to be locked in the TLBs. Each locked TLB entry reduces the number of TLB entries available to hold other translation information. The entries one would expect to lock in the TLBs are those used during access to locked cache lines. A TLB global invalidate does not affect locked entries.

During a context switch, an OS implementation may flush the TLBs. However, the OS is free to lock critical TLB entries in the TLBs reduce excessive thrashing and, therefore, retain performance.

3.4 Optimizing for Internal Memory Usage

The PXA27x processor has a 256-Kbyte memory that offers low latency and high memory bandwidth. Any data structure that requires high throughput and lower latency can be placed in the internal memory. While the LCD frame buffer is highly likely to be mapped to the internal memory—depending on the LCD size and refresh rate and latency that LCD can tolerate—some overlays can be placed in the external memory. This scheme can free up some internal memory space for OS and user applications. Depending on the user profile, the internal memory can be used for different purposes.

3.4.1 LCD Frame Buffer

The LCD is a significant bandwidth consumer in the system. The LCD frame buffer can be mapped to the internal memory. Apart from using the LCD frame buffer, the internal memory space may be used for an application frame buffer. Many applications update the image to be displayed in their local copy of frame buffer and then copy the content into the LCD frame buffer. Depending on the application update rate and LCD size, it might be preferable to let the application update the application frame buffer while system DMA can copy from the application frame buffer to the LCD frame buffer.

The LCD controller uses its DMA controller to fetch data from the frame buffer, which makes it possible to split the frame buffers between internal SRAM and external memory, if necessary, through the use of chained DMA descriptors. In this way, it is possible to use the internal SRAM for a portion of the frame buffer, even if the entire frame buffer cannot fit within the 256KB. For example, this could be useful for frame buffers with 640x480 (VGA) resolution.

3.4.2 Buffer for Capture Interface

The capture frames at the camera interface are processed typically for image enhancements and often encoded for transmission or storage. The image enhancement and video-encoding application is accelerated by allowing the storage of the raw data in the internal memory. Note that the capture interface can be on board or be an external device. Both benefit from the use of the internal-memory buffering scheme.

3.4.3 Buffer for Context Switch

During context switch, the states of the process have to be saved. For the PXA27x processor, the process control block (PCB) can be large in size due to additional registers for Intel® Wireless MMX™ Technology. To reduce context-switch latency, the internal memory can be used.

3.4.4 Scratch RAM

For many application (such as graphics), the working set may often be larger than the data cache, and due to the random access nature of the application effective preload may be difficult to perform. Thus, use part of the internal RAM for storing these critical data structures. The OS can offer management of such critical data spaces through *malloc()* or *virtual_alloc()*.

3.4.5 OS Acceleration

There is much OS- and system- related code that is used in a periodic fashion (for example, device drivers, OS daemon processes). Store code for these routines in the internal memory to reduce the instruction-cache miss penalties for the periodic routines.

3.4.6 Increasing Preloads for Memory Performance

Apart from increasing cache efficiency, hiding the memory latency is extremely important. The proper preload scheme can be used to hide the memory latency for data accesses.

The Intel XScale® Microarchitecture has a preload load instruction (**PLD**). The purpose of this instruction is to preload data into the data and mini-data caches. Data pre-loading allows hiding of memory-transfer latency while the processor continues to execute instructions. The preload is important to compiler and assembly code because judicious use of the preload instruction can improve enormously throughput performance of Intel XScale® Microarchitecture-based processors. Data preload can be applied not only to loops but also to any data references within a block of code. Preload also applies to data writing when the memory type is enabled as write allocate.

Note: The Intel XScale® Microarchitecture PLD instruction encoding translates to a “never execute” in the ARM* V4 architecture to allow compatibility between code using PLD on an Intel XScale® Microarchitecture processor and older devices. Code that has to run on both architectures can include the PLD instruction, gaining performance on the Intel XScale® Microarchitecture while maintaining compatibility for ARM* V4 (for example, StrongARM). A detailed discussion on the efficient pre-loading of the data and possible use cases has been explained in Section 4, “Intel XScale® Microarchitecture & Intel® Wireless MMX™ Technology Optimization”, Section 5, “High Level Language Optimization”, and Section 6, “Power Optimization”.

3.5 Optimization of System Components

In the PXA27x processor, the LCD, DMA controller, Intel® Quick Capture Interface and Intel XScale® core share the same resources such as system bus, memory controller, etc. Thus, there can be potential resource conflicts and the sharing of resources can impact the performance of the end application. For example, a larger LCD display consumes more memory and system bus bandwidth and therefore an application potentially could run faster in a system with a smaller LCD display or a display with a lower refresh rate. Also, DMA channels can influence the performance of applications. This section describes how to optimize different subsystems for improving system performance.

3.5.1 LCD Controller Optimization

The LCD controller provides an interface between the PXA27x processor and a LCD module. The LCD module can be passive (STN), active (TFT), or an LCD panel with internal frame buffering.

3.5.1.1 Bandwidth and Latency Requirements for LCD

The LCD controller can have up to seven DMA channels running, depending on the mode of operation. Therefore, the LCD potentially can consume the majority of the bus bandwidth when used with large panels. Consider bandwidth requirements for each plane (that is: base, overlay1, overlay 2, etc.) when determining LCD bandwidth requirements. The formula for each plane is:

$$\text{Plane Bandwidth} = \left(\frac{\text{Length} \times \text{Width} \times \text{Refresh Rate} \times \text{BPP}}{8}\right) \text{Bytes / Second}$$

"Length" and "width" are the number of lines per panel and pixels per line, respectively. "Refresh rate" is in frames per second. "BPP" is bits per pixel in physical memory: 16 for 16 BPP, 32 for 18 BPP unpacked, 24 for 18 BPP packed (refer to the *Intel® PXA27x Processor Family Developer's Manual* for more info).

Depending on where the overlay planes are placed, there might be a variable data bandwidth requirement during a refresh cycle of the LCD. The sections on the screen with overlaps between overlay 1 and 2 require fetching data at the highest rate. It is important to understand both the long term average and the peak bandwidth. The *average bandwidth* is a long-term average of the consumed bandwidth over the entire frame. The *peak bandwidth* is the highest (instantaneous) data rate that the LCD consumes, which occurs when fetching data for the overlapped section of the frame.

The average bandwidth can be calculated as:

$$\text{Average Bandwidth} = \sum(\text{Plane Bandwidths})$$

The formula for peak bandwidth is:

$$\text{Peak Bandwidth} = \text{Maximum Plane Overlap} \times \text{Base Plane Bandwidth}$$

"Maximum plane overlap" is the maximum number of overlapping planes (base, overlay 1, overlay 2). The planes do not need to completely overlap each other; they simply need to occupy the same pixel location. It is generally the number of planes used, unless the overlays are guaranteed never to be positioned over one another. The peak bandwidth is required whenever the LCD controller is displaying a portion of the screen where the planes overlap. While the peak bandwidth is higher than the average bandwidth, it does not sustain for long. Sustained period of peak bandwidth activity depends on the overlay sizes and color depth.

The system needs to guarantee the LCD has enough bandwidth available to meet peak bandwidth requirements for the sustained peak-bandwidth period to avoid underruns during plane-overlap periods. Optimizing arbitration scheme and internal-memory use is encouraged to address this problem. The LCD controller has an internal buffering mechanism to minimize the impact of fluctuations in the bandwidths.

The maximum latency the LCD controller can tolerate for its 32-byte burst data fetches can be calculated with the equation below. Note that the latency requirements can vary for different overlays (refer to Table 3-6, "Sample LCD Configurations with Latency and Peak Bandwidth Requirements").

$$\text{Latency} = \left(\frac{32}{\text{Peak Bandwidth}}\right) \text{ seconds}$$

Peak bandwidth comes from the equation above and is in bytes per second.

- So, for example, a 640x480x16 BPP screen with a 320x240x16 BPP overlay and a 70-Hz refresh rate, the average bandwidth required is:

  [(480 x 640 x 70 x 16) / 8] + [(240 x 320 x 70 x 16) / 8]

  = 43008000 + 10752000

  = 53,760,000 bytes per sec, or 52 MBytes/sec.

- The peak bandwidth required is:

  2 x [(480 x 640 x 70 x 16) / 8] = 86,016,000 bytes per sec, or 82 Mbytes per sec.

- The maximum allowable average latency for LCD DMA burst data fetches is:

  32 / 86,016,000 = 372 ns.

- For a 195 MHz system bus, this is $(372 \times 10^{-9}) \times (195 \times 10^{6}) = 72$ system bus cycles

Note that each LCD DMA channel has a 16-entry, 8-byte wide FIFO buffer to help deal with fluctuations in available bandwidth due to spikes in system activity.

Table 3-6. Sample LCD Configurations with Latency and Peak Bandwidth Requirements

LCD (Base Plane Size, Overlay 1, Overlay 2, Cursor)	Refresh Rate (Hz)	Color Depth	Frame Buffer Foot Print Requirement (KBytes)	Maximum Latency Tolerance (ns)	Average Bandwidth Requirements (MBytes/Sec)	Peak Bandwidth Requirements (MBytes / sec)
320x240+ No Overlay	77	16 BPP	150	2702.56	11.28	11.28
640x480 + No Overlay	78	18 BPP unpacked	1200	333.87	91.41	91.41
640x480 + No Overlay	78	18 BPP packed	900	445.16	68.55	68.55
800x600+ No Overlay	73	16 BPP	937.5	456.62	66.83	66.83
800x600 + 176x144 Overlay	73	16 BPP	937.5 base + 49.5 overlay	223.78	70.52	133.67

3.5.1.2 Frame Buffer Placement for LCD Optimization

3.5.1.2.1 Internal Memory Usage

As the bandwidth and latency requirements increase with screen size, it may become necessary to use internal memory to meet LCD requirements. Internal memory provides the lowest latency and highest bandwidth of all memories in the system. In addition, having the frame buffer located in internal SRAM dramatically reduces the external memory traffic in the system and the internal bus-utilization.

3.5.1.2.2 Overlay Placement

Most systems that use overlays require more memory for the frame buffers (base plane and overlays) than is available (or allocated for frame buffer usage) in the internal SRAM. Optimum system performance is achieved by placing the most frequently accessed frame buffers in internal SRAM and placing the remainder in external memory. Frame buffer accesses include not only the OS and applications writing to the plane when updating the content displayed, but also the LCD controller reading the data from the plane.

For the base plane, the total accesses are simply the sum of the refresh rate plus the frequency of content update of the base plane. For each overlay, the total accesses are the same sum multiplied by the percent of time the overlay is enabled. After estimating the total accesses for the base plane and all overlays employed, place the frame buffers for the planes with the highest total accesses in the internal SRAM.

Some systems might benefit from dynamically reconfiguring the location of the frame buffer memory whenever the overlays are enabled. When overlays are disabled, the frame buffer for the base plane is placed in the internal SRAM. However, when the overlays are enabled, the base plane frame buffer is moved to external memory and the frame buffers of the overlays are placed in the internal SRAM. This method requires close coordination with the LCD controller to ensure that no artifacts are seen on the LCD. Refer to the LCD chapter in the *Intel® PXA27x Processor Family Developer's Manual* for more information on reconfiguring the LCD.

3.5.1.2.3 Multiple Descriptor Technique

Another technique for using internal SRAM is the use of multiple-descriptor frames. This technique can be used if the LCD controller underruns due to occasional slow memory accesses. The frame buffer is split across two different chained descriptors for a particular LCD DMA channel. Both descriptors can be stored in internal SRAM for speedy descriptor reloading. One descriptor points to frame buffer source data in external memory, while the second descriptor points to the remainder of the frame buffer in internal SRAM. The descriptor that points to frame data in external memory should have the end-of-frame (EOF) interrupt bit set. The idea is to queue slower memory transfers and push them out when the EOF interrupt occurs, indicating that the LCD is switching to internal SRAM. This allows a ping-pong between slow memory traffic (that would cause LCD output FIFO underruns) and LCD traffic. This technique is necessary only with very large screens, and does not work if the offending slow memory accesses also occurs when the LCD is fetching from external memory.

3.5.1.3 LCD Display Frame Buffer Settings

For most products the LCD frame buffer is allocated in a noncacheable region. If this region is set to noncached but bufferable, graphics performance improvements can be achieved. The noncached but bufferable mode (X=0, C=0, B=1) improves write performance by allowing consecutive writes to coalesce in the write buffer and results in more efficient bus transactions. Set the LCD frame buffer as noncached but bufferable.

Another potential technique is to map the region occupied by the frame buffer into 1-MB sections instead of 4-kB or 64-kB pages. This technique reduces the number of page-table translations that occur. See Section 3.3.3, "Optimizing TLB (Translation Lookaside Buffer) Usage" for more information.

3.5.1.4 LCD Color-Conversion Hardware

In video and image applications, the color-conversion routines require a significant amount of processing power. This work can be off-loaded to the LCD controller by properly configuring the LCD controller as described below. This offloading offers two advantages: first, the Intel XScale® core is not burdened with the processing, and second, the LCD bandwidth consumption is lowered by using the lower bit-precision format.

Overlay 2 supports the following formats and pixel depths:

- RGB format, pixel depths of 4, 8, 16, 18, 19, 24, and 25 bpp
- 4:4:4 YCbCr sampling format
- 4:2:2 YCbCr sampling format
- 4:2:0 YCbCr sampling format.

The LCD controller Overlay 2 has hardware color-management capabilities to perform the following chain of sampling and conversion functions:

- Up-scaling from YCbCr 4:2:0 & 4:2:2 to YCbCr 4:4:4
- Color Space Conversion from YCbCr 4:4:4 to RGB 8:8:8 (CCIR 601)
- Conversion from RGB 8:8:8 to RGB 5:5:5 and the supported formats of RGBT

Use Overlay 2 to perform color conversion to help video performance. For example, the output of many MPEG-4 video decoders is YCbCr 4:2:0, which can be converted to RGBT 5:5:5 16-bit-per-pixel format to be displayed on an LCD panel.

3.5.1.5 Arbitration Scheme Tuning for LCD

The most important task to enable larger screens is to reprogram the arbiter ARBCTRL register. The default arbiter weight for the programmable clients is LCD=2, DMA=3 and Intel XScale® core=4; this is sufficient only for very small screens. Typically, the LCD needs to be the highest weight of the programmable clients - this is discussed further in Section 3.5.2, "Optimizing Arbiter Settings".

3.5.2 Optimizing Arbiter Settings

3.5.2.1 Arbiter Functionality

The PXA27x processor arbiter features programmable "weights" for the LCD controller, DMA controller, and Intel XScale® Microarchitecture bus requests. In addition, the "park" bit can be set, which causes the arbiter to grant the bus to a specific client whenever the bus is idle. Use these two features to tune the PXA27x processor to match the system-bandwidth requirements.

The USB host controller cannot tolerate long latencies and is given highest priority whenever it requests the bus, unless the memory controller is requesting the bus. The memory controller has the absolute highest priority in the system. Since the weights of the USB host and memory controller are not programmable, they are not discussed any further.

The weights of the LCD, DMA controller and Intel XScale® Microarchitecture bus requests are programmable via the ARBCNTL register. The maximum weight allowed is 15. Each client weight is loaded into a counter, and whenever a client is granted the bus, the counter decrements. When all counters reach zero, the counters are reloaded with the weights in the ARBCNTL register and the process restarts. At any given time, the arbiter gives a grant to the client with the highest value in their respective counter, unless the USB host or memory controller is requesting the bus. If one or more client counts are at zero and no non-zero clients are requesting the bus, the arbiter grants the bus to the zero-count client with the oldest pending request. If this happens three times, the counters are all reloaded even though one more client counts never reached zero.

This basic understanding of how the arbiter works is necessary to begin tuning the arbiter settings.

3.5.2.2 Determining the Optimal Weights for Clients

The weights are decided based on the real time (RT) deadline[1], bandwidth (BW) requirements and likelihood of a client requesting the bus. Setting the correct weight helps ensure that each client is statistically guaranteed to have a fixed amount of bandwidth.

Overassigning or underassigning weights may violate the BW and RT requirements of a client. When weights for one or more clients becomes zero, the effective arbitration becomes first-come, first-serve (FCFS).

3.5.2.2.1 Weight for LCD

The first client to consider is the LCD controller. When used with larger panel sizes or overlays, the LCD controller has very demanding real-time data requirements, which if not satisfied, results in underruns and visual artifacts. Therefore, the LCD controller is usually given the highest weight of all of the programmable clients. The safest and easiest method of ensuring the LCD controller gets all of the bandwidth it requires is to set the LCD weight to 15. This weight gives the LCD controller the bus whenever it needs it, allowing the LCD FIFO buffers to stay as full as possible to avoid underrun situations. The remaining bus bandwidth, which may be very little if a very large panel is used, is then split up between the DMA controller and the Intel XScale® Microarchitecture.

1. Real-time deadline is the maximum time that a client can wait for data across the bus without impacting the client's performance (for example, by causing a stall).

3.5.2.2.2 Weight for DMA

The DMA controller is a unique client in that it is “friendly” and always de-asserts its request line whenever it receives a grant. Therefore, it only performs back-to-back transactions if no other device is requesting the bus. In addition, if the DMA controller is the only non-zero client, there is a fair chance the client counters are prematurely reloaded due to three zero-count clients getting grants in between DMA grants. For these reasons, the DMA controller never consumes all of the bus bandwidth, even when programmed with a large weight. The best weight to use depends on the system, based on the number of DMA channels running and the bandwidth requirements of those channels. Because the LCD controller and USB host have real-time requirements, DMA bandwidth usually reduces the available bandwidth to the Intel XScale® core.

3.5.2.2.3 Weight for Core

A good method for setting the Intel XScale® core weight and the DMA controller weight is to determine the ratio of the bandwidth requirements of both. Once the ratio is determined, the weights can be programmed with that same ratio. For instance, if the Intel XScale® core requires twice the bandwidth of the DMA controller, the DMA weight could be set to two with the Intel XScale® core weight set to four. Larger weights are used for greater accuracy, but the worst-case time to grant also increases. It is often best to start with low weights while the LCD weight is high to avoid LCD problems at this point. The weights can be increased using the same ratio, if preferred, and there are no LCD underruns.

3.5.2.3 Taking Advantage of Bus Parking

Another arbiter feature is the ability to park the grant on a particular client when the bus is idle. If the bus is not parked and is idle, it takes 1-2 cycles to obtain a bus grant. This can be reduced to zero if the bus is successfully parked on the next client that needs it. However, if the bus is parked on a particular client and a different client requests the bus, it takes 2-3 cycles to receive a bus grant. This results in a 1-cycle penalty for a mispredicted park.

For most applications with smaller LCD screen resolutions, Intel recommends parking the bus on the core. Since the bus parking can be changed easily and dynamically, Intel also recommends that the OS and applications use this feature to park the bus where it results in the best performance for the current task.

While most applications have the highest performance with the bus parked on the Intel XScale® core, some might perform better with different bus park settings. For example, systems with larger (VGA) LCD resolutions may benefit from parking on the LCD to avoid underruns. For another example, it is likely that parking the bus on the memory controller results in higher performance than having it parked on the core if an application was invoked to copy a large section of memory from SDRAM. Use the performance monitoring capabilities of the Intel XScale® core to verify that the choice of bus parking resulted in increased performance.

3.5.2.4 Dynamic Adaptation of Weights

Once the initial weights for all of the programmable clients have been determined, test the arbiter settings with real system traffic. It is important to ensure all real-time requirements are met with both typical and worst-case traffic loads. It may require several iterations to find the best arbiter setting. Once the best ratio is determined, increase arbiter accuracy by raising the DMA controller and Intel XScale® core weights as much as possible while still preserving the ratio between the two. Retest the system while increasing weights to ensure the increase in worst-case time-to-grant does not affect performance. Also, monitor LCD output FIFO buffer underruns to ensure the LCD

does not fail as its bandwidth allocation decreases. If worst-case time-to-grant is more important than arbiter accuracy, use smaller weights and lower the LCD weight as long as LCD output FIFO underruns do not occur with a worst-case traffic load.

A final consideration is dynamically changing the ARBCNTL register based on the current state of the system. For example, experimentation may show different DMA controller weights should be used based on the number of channels running. When the system enables a new channel, the ARBCNTL register can be written, which results in an immediate reload of the client counters.

3.5.3 Usage of DMA

The DMA controller is used by the PXA27x processor peripherals for data transfers between the peripheral buffers and the memory (internal and external). Also, depending on the use cases and user profiles, the operating system may use DMA for copying different pages for its own operations. Table 3-7 shows DMA controller performance data.

Table 3-7. Memory to Memory Performance Using DMA for Different Memories and Frequencies

Clock Ratios†	DMA Throughput for Internal to Internal Memory (MB/sec)	DMA Throughput for Internal to External Memory (MB/sec)
104:104:104	127.3	52.9
208:104:104	127.6	52.3
195:195:97.5	238.2	70.9
338:169:84.5	206	59.4
390:195:97.5	237.9	68.6

† Ratio = Core Frequency : System Bus Frequency : Memory Bus Frequency

Proper DMA controller use can reduce the processor workload by allowing the Intel XScale® core to use the DMA controller to perform peripheral I/O. The DMA can also be used to populate the internal memory from the capture interface or external memory, etc.

3.5.4 Peripheral Bus Split Transactions

The DMA bridge between the peripheral bus and the system bus normally performs split transactions for all operations. These transactions allow for some decoupling of the address and data phases of transactions and generally improve efficiency. This can be disabled and requires active transactions complete before another transaction starts. However, disabling split-transactions may reduce in reduced memory performance. Refer to the DMA Programmed I/O Control Status register described in the *Intel® PXA27x Processor Family Developer's Manual* for detailed information on this feature and its usage.

Note: When using split transactions (default): If software requires that a write complete on the peripheral bus before continuing, software must write the address, then immediately read the same address. This guarantees that the address has been updated before letting the core continue execution. The user must perform this read-after-write transaction to ensure the processor is in a correct state before the core continues execution.

Intel XScale® Microarchitecture and Intel® Wireless MMX™ Technology Optimization 4

4.1 Introduction

This section outlines optimizations specific to ARM* architecture and also to the Intel® Wireless MMX™ Technology. These optimizations are modified for the Intel XScale® Microarchitecture where needed. This chapter focuses mainly on the assembly code level optimization. Chapter 5 explains optimization during high level language programming.

4.2 General Optimization Techniques

The Intel XScale® Microarchitecture provides the ability to execute instructions conditionally. This feature combined with the ability of the Intel XScale® Microarchitecture instructions to modify the condition codes makes a wide array of optimizations possible.

4.2.1 Conditional Instructions and Loop Control

The Intel XScale® Microarchitecture instructions can selectively modify the state of the condition codes. When generating code for *if-else* and *loop* conditions it is often beneficial to make use of this feature to set condition codes, thereby eliminating the need for a subsequent compare instruction.

Consider the following C statement

```
if (a + b)
```

Code generated for the *if* condition without using an add instruction to set condition codes is:

```
; Assume r0 contains the value a, r1 contains the value b,
; and r2 is available

add   r2,r0,r1
cmp   r2, #0
```

However, code can be optimized as follows making use of add instruction to set condition codes:

```
; Assume r0 contains the value a, r1 contains the value b,
; and r2 is available
```

```
adds   r2,r0,r1
```

The instructions that increment or decrement the loop counter can also modify the condition codes. This eliminates the need for a subsequent compare instruction. A conditional branch instruction can then exit or continue with the next loop iteration.

Consider the following C code segment.

```
for (i = 10; i!= 0; i--){
   do something;
}
```

The optimized code generated for the preceding code segment would look like:

```
L6:
      subs r3, r3, #1
      bne  .L6
```

It is also beneficial to rewrite loops whenever possible to make the loop-exit conditions check against the value 0. For example, the code generated for the code segment below needs a compare instruction to check for the loop-exit condition:

```
 for (i = 0; i < 10; i++){
    do something;
}
```

If the loop were rewritten as follows, the code generated avoids using the compare instruction to check for the loop-exit condition:

```
 for (i = 9; i >= 0; i--){
    do something;
}
```

4.2.2 Program Flow and Branch Instructions

Branches decrease application performance by indirectly causing pipeline stalls. Branch prediction improves performance by lessening the delay inherent to fetching a new instruction stream. The Intel® PXA27x Processor Family (PXA27x processor) add a branch target buffer (BTB), which helps mitigate the penalty due to branch misprediction. However, the BTB must be enabled.

The size of the branch target buffer limits the number of correctly predictable branches. It is often beneficial to minimize the number of branches in a program because the total number of branches executed in a program is relatively large compared to the size of the branch target buffer. Consider the following C code segment:

```
 int foo(int a) {
```

```
        if (a > 10)
           return 0;
        else
           return 1;
    }
```

The code generated for the *if-else* portion of this code segment using branches is:

```
    cmp   r0, #10
    ble   L1
    mov   r0, #0
    b     L2
L1:
    mov   r0, #1
L2:
```

This code takes three cycles to execute the *else* statement and four cycles for the *if* statement assuming best case conditions and no branch misprediction penalties. In the case of the Intel XScale® Microarchitecture, a branch misprediction incurs a penalty of four cycles. On average, the code above takes 5.5 cycles to execute when the branch is mispredicted 50% of the time and when both the *if* statement and the *else* statement are equally likely to be taken.

$$\left(\frac{50}{100}\times 4+\frac{3+4}{2}\right) = 5.5 \qquad cycles\,.$$

Using the Intel XScale® Microarchitecture to execute instructions conditionally, the code generated for the preceding *if-else* statement is:

```
    cmp   r0, #10
    movgt r0, #0
    movle r0, #1
```

The preceding code segment would not incur any branch misprediction penalties and would take three cycles to execute assuming best case conditions. Using conditional instructions speeds up execution significantly. However, carefully consider the use of conditional instructions to ensure they improve performance. To decide when to use conditional instructions over branches, consider this hypothetical code segment:

```
if (cond)
   if_stmt
else
   else_stmt
```

Using the following data:

$N1_B$ Number of cycles to execute the if_stmt assuming the use of branch instructions

$N2_B$ Number of cycles to execute the else_stmt assuming the use of branch instructions

P1 Percentage of times the if_stmt is likely to be executed

P2 Percentage of times likely to incur a branch misprediction penalty

$N1_C$ Number of cycles to execute the *if-else* portion using conditional instructions assuming the *if*-condition to be true

$N2_C$ Number of cycles to execute the *if-else* portion using conditional instructions assuming the *if*-condition to be false

Use conditional instructions when:

$$\left(N1_C \times \frac{P1}{100}\right) + \left(N2_C \times \frac{100 - P1}{100}\right) \leq \left(N1_B \times \frac{P1}{100}\right) + \left(N2_B \times \frac{100 - P1}{100}\right) + \left(\frac{P2}{100} \times 4\right)$$

The following example illustrates a situation in which it is better to use branches instead of conditional instructions.

```
    cmp   r0, #0
    bne   L1
    add   r0, r0, #1
    add   r1, r1, #1
    add   r2, r2, #1
    add   r3, r3, #1
    add   r4, r4, #1
    b     L2
L1:
    sub   r0, r0, #1
    sub   r1, r1, #1
    sub   r2, r2, #1
    sub   r3, r3, #1
    sub   r4, r4, #1
L2:
```

The CMP instruction takes one cycle to execute, the *if* statement takes seven cycles to execute, and the *else* statement takes six cycles to execute.

If the code were changed to eliminate the branch instructions by using *of* conditional instructions, the *if-else* statement would take 10 cycles to complete.

Assuming an equal probability of both paths being taken and that branch misprediction occur 50% of the time, compute the costs of branch prediction versus conditional execution as:

- Cost of using conditional instructions:

$$1 + \left(\frac{50}{100} \times 10\right) + \left(\frac{50}{100} \times 10\right) = 11 \qquad cycles$$

- Cost of using branches:

$$1 + \left(\frac{50}{100} \times 7\right) + \left(\frac{50}{100} \times 6\right) + \left(\frac{50}{100} \times 4\right) = 9.5 \qquad cycles$$

Users get better performance by using branch instructions in this scenario.

4.2.3 Optimizing Complex Expressions

Using conditional instructions improves the code generated for complex expressions such as the C shortcut evaluation feature. The use of conditional instructions in this fashion improves performance by minimizing the number of branches, thereby minimizing the penalties caused by branch misprediction.

```
int foo(int a, int b){
   if (a!= 0 && b!= 0)
      return 0;
   else
      return 1;
   }
```

The optimized code for the *if* condition is:

```
cmp   r0, #0
cmpne r1, #0
```

Similarly, the code generated for this C segment:

```
int foo(int a, int b){
   if (a!= 0 || b!= 0)
      return 0;
   else
      return 1;
   }
```

is:

```
cmp   r0, #0
cmpeq r1, #0
```

This approach also reduces the utilization of branch prediction resources.

4.2.3.1 Bit Field Manipulation

The Intel XScale® Microarchitecture shift and logical operations provide a useful way of manipulating bit fields. Bit field operations can be optimized as:

```
;Set the bit number specified by r1 in register r0
 mov   r2, #1
 orr   r0, r0, r2, asl r1

;Clear the bit number specified by r1 in register r0
 mov   r2, #1
 bic   r0, r0, r2, asl r1

;Extract the bit-value of the bit number specified by r1 of the
;value in r0 storing the value in r0
 mov   r1, r0, asr r1
 and   r0, r1, #1

;Extract the higher order 8 bits of the value in r0 storing
;the result in r1
 mov   r1, r0, lsr #24
```

The method outlined here can greatly accelerate encryption algorithms such as Data Encryption Standard (DES), Triple DES (T-DES), Hashing functions (SHA). This approach helps other application such as network-packet parsing and voice-stream parsing.

4.2.4 Optimizing the Use of Immediate Values

Use the Intel XScale® Microarchitecture MOV or MVN instruction when loading an immediate (constant) value into a register. Refer to the *ARM* Architecture Reference Manual* for the set of immediate values that can be used in a MOV or MVN instruction. It is also possible to generate a whole set of constant values using a combination of MOV, MVN, ORR, BIC, and ADD instructions. The LDR instruction has the potential of incurring a cache miss in addition to polluting the data and instruction caches. Use a combination of the above instructions to set a register to a constant value. An example of this is shown in these code samples.

```
;Set the value of r0 to 127
 mov   r0, #127

;Set the value of r0 to 0xfffffefb.
 mvn   r0, #260

;Set the value of r0 to 257
 mov   r0, #1
 orr   r0, r0, #256

;Set the value of r0 to 0x51f
 mov   r0, #0x1f
 orr   r0, r0, #0x500
```

```
;Set the value of r0 to 0xf100ffff
 mvn   r0, #0xff, LSL 16
 bic   r0, r0, #0xe, LSL 8

; Set the value of r0 to 0x12341234
 mov   r0, #0x8d, LSL 2
 orr   r0, r0, #0x1, LSL 12
 add   r0, r0, r0, LSL #16
; shifter delay of 1 cycle
```

It is possible to load any 32-bit value into a register using a sequence of four instructions.

4.2.5 Optimizing Integer Multiply and Divide

Optimize when multiplying by an integer constant to make use of the shift operation.

```
;Multiplication of R0 by 2^n
 mov   r0, r0, LSL #n

;Multiplication of R0 by 2^n+1
 add   r0, r0, r0, LSL #n
```

Multiplication by an integer constant, expressed as $(2^n + 1) \cdot (2^m)$, can be optimized.

```
;Multiplication of r0 by an integer constant that can be
;expressed as (2^n+1)*(2^m)
 add   r0, r0, r0, LSL #n
 mov   r0, r0, LSL #m
```

Note: Use the preceding optimization in cases where the multiply operation cannot be advanced far enough to prevent pipeline stalls only.

Optimize when dividing an unsigned integer by an integer constant to make use of the shift operation.

```
;Dividing r0 containing an unsigned value by an integer constant
;that can be represented as 2^n
 mov   r0, r0, LSR #n
```

Optimize when dividing a signed integer by an integer constant to make use of the shift operation.

```
;Dividing r0 containing a signed value by an integer constant
;that can be represented as 2^n
 mov   r1, r0, ASR #31
 add   r0, r0, r1, LSR #(32 - n)
 mov   r0, r0, ASR #n
```

The add instruction stalls for one cycle. Prevent this stall by filling in another instruction before the add instruction.

4.2.6 Effective Use of Addressing Modes

The Intel XScale® Microarchitecture provides a variety of addressing modes that make indexing an array of objects highly efficient. Refer to the *ARM* Architecture Reference Manual* for a detailed description of ARM*addressing modes. These code samples illustrate how various kinds of array operations can be optimized to make use of the various addressing modes:

```
;Set the contents of the word pointed to by r0 to the value
;contained in r1 and make r0 point to the next word
 str   r1,[r0], #4

;Increment the contents of r0 to make it point to the next word
;and set the contents of the word pointed to the value contained
;in r1
 str   r1, [r0, #4]!

;Set the contents of the word pointed to by r0 to the value
;contained in r1 and make r0 point to the previous word
 str   r1,[r0], #-4

;Decrement the contents of r0 to make it point to the previous
;word and set the contents of the word pointed to the value
;contained in r1
 str   r1,[r0, #-4]!
```

4.3 Instruction Scheduling for Intel XScale® Microarchitecture and Intel® Wireless MMX™ Technology

This section discusses instruction scheduling optimizations. Instruction scheduling refers to the rearrangement of a sequence of instructions for the purpose of helping to minimize pipeline stalls. Reducing the number of pipeline stalls helps improve application performance while these rearrangements ensure the new sequence of instructions has the same effect as the original sequence of instructions.

4.3.1 Instruction Scheduling for Intel XScale® Microarchitecture

4.3.1.1 Scheduling Loads

On the Intel XScale® Microarchitecture, an LDR instruction has a result latency of 3 cycles, assuming the data being loaded is in the data cache. If the instruction after the LDR needs to use the result of the load, then it would stall for 2 cycles. If possible, rearrange the instructions surrounding the LDR instruction to avoid this stall.

In the code shown in the following example, the ADD instruction following the LDR stalls for two cycles because it uses the result of the load:

```
add   r1, r2, r3
ldr   r0, [r5]
add   r6, r0, r1
sub   r8, r2, r3
mul   r9, r2, r3
```

Rearrange the code as shown to prevent the stall:

```
ldr   r0, [r5]
add   r1, r2, r3
sub   r8, r2, r3
add   r6, r0, r1
mul   r9, r2, r3
```

This rearrangement is not always possible. In the following example, the LDR instruction cannot be moved before the ADDNE or the SUBEQ instructions because the LDR instruction depends on the result of these instructions:

```
cmp   r1, #0
addne r4, r5, #4
subeq r4, r5, #4
ldr   r0, [r4]
cmp   r0, #10
```

This example rewrites this code to make it run faster at the expense of increasing code size:

```
cmp   r1, #0
ldrne r0, [r5, #4]
ldreq r0, [r5, #-4]
addne r4, r5, #4
subeq r4, r5, #4
cmp   r0, #10
```

The optimized code takes six cycles to execute compared to the seven cycles required by the unoptimized version.

The result latency for an LDR instruction is significantly higher if the data being loaded is not in the data cache. To help minimize the number of pipeline stalls in such a situation, move the LDR instruction as far away as possible from the instruction that uses the result of the load. Moving the LDR instruction can cause certain register values to be spilled to memory due to the increase in register pressure. In such cases, use a preload instruction to ensure that the data access in the LDR instruction hits the cache when it executes.

In the following code sample, the ADD and LDR instructions can be moved before the MOV instruction to help prevent pipeline stalls if the load hits the data cache. However, if the load is likely to miss the data cache, move the LDR instruction so it executes as early as possible—before the SUB instruction. Moving the LDR instruction before the SUB instruction changes the program semantics.

```
; all other registers are in use
 sub   r1, r6, r7
 mul   r3,r6, r2
 mov   r2, r2, LSL #2
 orr   r9, r9, #0xf
 add   r0,r4, r5
 ldr   r6, [r0]
 add   r8, r6, r8
 add   r8, r8, #4
 orr   r8,r8, #0xf
; The value in register r6 is not used after this
```

It is possible to move the ADD and the LDR instructions before the SUB instruction so that the contents of register R6 are allowed to spill and restore from the stack as shown in this example:

```
; all other registers are in use
 str   r6,[sp, #-4]!
 add   r0,r4,r5
 ldr   r6, [r0]
 mov   r2, r2, LSL #2
 orr   r9, r9, #0xf
 add   r8, r6, r8
 ldr   r6, [sp], #4
 add   r8, r8, #4
 orr   r8,r8, #0xf
 sub   r1, r6, r7
 mul   r3,r6, r2
; The value in register R6 is not used after this
```

In the previous example, the contents of register R6 are spilled to the stack and subsequently loaded back to register R6 to retain the program semantics. Using a preload instruction, such as the one shown in the following example, is another way to optimize the code in the previous example.

```
; all other registers are in use
 add   r0,r4, r5
 pld   [r0]
 sub   r1, r6, r7
 mul   r3,r6, r2
 mov   r2, r2, LSL #2
 orr   r9, r9, #0xf
 ldr   r6, [r0]
 add   r8, r6, r8
```

```
    add   r8, r8, #4
    orr   r8,r8, #0xf
   ; The value in register r6 is not used after this
```

The Intel XScale® Microarchitecture has four fill buffers used to fetch data from external memory when a data cache miss occurs. The Intel XScale® Microarchitecture stalls when all fill buffers are in use. This happens when more than four loads are outstanding and are being fetched from memory. Write the code to ensure no more than four loads are simultaneously outstanding. For example, the number of loads issued sequentially should not exceed four. A preload instruction can cause a fill buffer to be used. As a result, the number of outstanding preload instructions should also be considered to arrive at the number of loads that are outstanding.

Use the number of outstanding loads to improve performance of the PXA27x processor.

4.3.1.2 Increasing Load Throughput

Increasing load throughput for data-demanding applications is important. Making use of multiple outstanding loads increases throughput in the PXA27x processor. Use register rotation to allow multiple outstanding loads. The following code allows one outstanding load at a time due to the data dependency between the instructions (load and add). Throughput falls drastically in cases where there is a cache miss.

```
Loop:
   ldr r1, [r0], #32; r0 be a pointer to some initialized memory
   add r2, r2, r1
   ldr r1, [r0], #32;
   add r2, r2, r1
   ldr r1, [r0], #32;
   add r2, r2, r1
  .
  .
  .
   bne Loop
```

However, the following example uses multiple registers as the target for loads and allows multiple outstanding loads:

```
   ldr r1, [r0], #32; r0 be a pointer to some initialized memory
   ldr r2, [r0], #32
   ldr r3, [r0], #32
   ldr r4, [r0], #32

Loop:
   add r5, r5, r1
   ldr r1, [r0], #32
   add r5, r5, r3
   ldr r2, [r0], #32
   add r5, r5, r3
   ldr r3, [r0], #32
   add r5, r5, r4
```

```
ldr r4, [r0], #32
.
.
.
bne Loop
```

The modified code not only hides the load-to-use latencies for the cases of cache-hits, but also increases the throughput by allowing several loads to be outstanding at a time.

Due to the complexity of the PXA27x processor, the memory latency can be higher. Latency hiding is very critical. Thus, two things to remember: issue loads as early as possible, and make up to four outstanding loads. Another technique for hiding memory latency is to use preloads. The prefetch technique is mentioned in Chapter 5, "High Level Language Optimization".

4.3.1.3 Increasing Store Throughput

Increasing store throughput is important in applications that process video while updating the output to the display. Write coalescing in the PXA27x processor (set by the page table attributes) combines multiple stores going to the same half of the cache line into a single memory transaction. This approach increases the bus efficiency and throughput. The coalescing operation is transparent to software. However, software can cause more frequent coalescing by placing store instructions targeted to the same cache line next to each other and configuring the target-page attributes as bufferable. For example, this code does not take advantage of coalescing:

```
add r1, r1,r2
str r1,[r0],#4 ; A separate bus transaction
add r1, r1,r3
str r1,[r0],#4; A separate bus transaction
add r1, r1,r4
str r1,[r0],#4; A separate bus transaction
add r1, r1,r5
str r1,[r0],#4; A separate bus transaction
```

However, it can be modified to allow coalescing to occur as:

```
add r1, r1,r2
add r6, r1,r3
add r7, r6,r4
add r8, r7,r5
str r1,[r0],#4
str r6,[r0],#4
str r7,[r0],#4
str r8,[r0],#4; All four writes can now coalesce into one trans.
```

Section 4.6 contains case studies showing typical functions such as memory fill and zero fill, and their acceleration with coalescing.

The number of write buffers limits the number of successive writes that can be issued before the processor stalls. No more than eight uncoalesced store instructions can be issued. If the data caches are using the write-allocate with writeback policy, then a load operation may cause stores to the external memory if the read operation evicts a cache line that is dirty (modified). The number of sequential stores may be limited by this fact.

4.3.1.4 Scheduling Load Double and Store Double (LDRD/STRD)

The Intel XScale® Microarchitecture introduces two new double word instructions: LDRD and STRD. LDRD loads 64 bits of data from an effective address into two consecutive registers. STRD stores 64 bits from two consecutive registers to an effective address. There are two important restrictions on how these instructions are used:

- The effective address must be aligned on an 8-byte boundary
- The specified register must be even (r0, r2)

Using LDRD/STRD instead of LDM/STM to do the same thing is more efficient because LDRD/STRD issues in only one or two clock cycles. LDM/STM issues in four clock cycles. Avoid LDRDs targeting R12 because this incurs an extra cycle of issue latency.

The LDRD instruction has a result latency of three or four cycles depending on the destination register being accessed (assuming the data being loaded is in the data cache).

```
 add   r6, r7, r8
 sub   r5, r6, r9
; The following ldrd instruction would load values
; into registers r0 and r1
 ldrd r0, [r3]
 orr   r8, r1, #0xf
 mul   r7, r0, r7
```

In the code example above, the ORR instruction stalls for three cycles because of the four-cycle result latency for the second destination register of an LDRD instruction. The preceding code can be rearranged to help remove the pipeline stalls:

```
; The following ldrd instruction would load values
; into registers r0 and r1
 ldrd r0, [r3]
 add   r6, r7, r8
 sub   r5, r6, r9
 mul   r7, r0, r7
 orr   r8, r1, #0xf
```

Any memory operation following a LDRD instruction (LDR, LDRD, STR and others) stall for one cycle.

```
; The str instruction below will stall for 1 cycle
 ldrd r0, [r3]
 str   r4, [r5]
```

4.3.1.5 Scheduling Load and Store Multiple (LDM/STM)

LDM and STM instructions have an issue latency of 2 to 20 cycles depending on the number of registers being loaded or stored. The issue latency is typically two cycles plus an additional cycle for each of the registers loaded or stored assuming a data cache hit. The instruction following an LDM stalls whether or not this instruction depends on the results of the load. An LDRD or STRD instruction does not suffer from this drawback (except when followed by a memory operation) and should be used where possible. Consider the task of adding two 64-bit integer values. Assume that the addresses of these values are aligned on an 8-byte boundary. Achieve this using the following LDM instructions.

```
; r0 contains the address of the value being copied
; r1 contains the address of the destination location
 ldm   r0, {r2, r3}
 ldm   r1, {r4, r5}
 adds r0, r2, r4
 adc   r1,r3, r5
```

Assuming all accesses hit the cache, this example code takes 11 cycles to complete. Rewriting the code as shown in the following example using the LDRD instruction would take only seven cycles to complete. The performance increases further if users fill in other instructions after the LDRD instruction to reduce the stalls due to the result latencies of the LDRD instructions and the one cycle stall of any memory operation.

```
; r0 contains the address of the value being copied
; r1 contains the address of the destination location
 ldrd  r2, [r0]
 ldrd  r4, [r1]
 adds  r0, r2, r4
 adc   r1,r3, r5
```

Similarly, the code sequence in the following example takes five cycles to complete:

```
 stm   r0, {r2, r3}
 add   r1, r1, #1
```

The alternative version which is shown below would only take 3 cycles to complete:

```
 strd  r2, [r0]
 add   r1, r1, #1
```

4.3.1.6 Scheduling Data-Processing

Most Intel XScale® Microarchitecture data-processing instructions have a result latency of one cycle. This means that the current instruction uses the result from the previous data processing instruction. However, the result latency is two cycles if the current instruction uses the result of the previous data processing instruction for a shift by immediate. As a result, this code segment would incur a one-cycle stall for the MOV instruction:

```
sub   r6, r7, r8
add   r1, r2, r3
mov   r4, r1, LSL #2
```

This code removes the one-cycle stall:

```
add   r1, r2, r3
sub   r6, r7, r8
mov   r4, r1, LSL #2
```

All data-processing instructions incur a two-cycle issue penalty and a two-cycle result penalty when the shifter operand is shifted/rotated by a register or the shifter operand is a register. The next instruction incur a two-cycle issue penalty; there is no way to avoid such a stall except by rewriting the assembler instruction. The subtract instruction incurs a one-cycle stall due to the issue latency of the add instruction as the shifter operand is shifted by a register.

```
mov   r3, #10
mul   r4, r2, r3
add   r5, r6, r2, LSL r3
sub   r7, r8, r2
```

The issue latency can be avoided by changing the code as:

```
mov   r3, #10
mul   r4, r2, r3
add   r5, r6, r2, LSL #10
sub   r7, r8, r2
```

4.3.1.7 Scheduling Multiply Instructions

Multiply instructions can cause pipeline stalls due to resource conflicts or result latencies. This code segment incurs a stall of 0-3 cycles depending on the values in registers R1, R2, R4 and R5 due to resource conflicts:

```
mul   r0, r1, r2
mul   r3, r4, r5
```

Due to result latency, this code segment incurs a stall of 1-3 cycles depending on the values in registers R1 and R2:

```
mul   r0, r1, r2
mov   r4, r0
```

A multiply instruction that sets the condition codes blocks the whole pipeline. A four-cycle multiply operation that sets the condition codes behaves the same as a four-cycle issue operation. The add operation in the following example stalls for three cycles if the multiply takes three cycles to complete.

```
muls  r0, r1, r2
add   r3, r3, #1
sub   r4, r4, #1
sub   r5, r5, #1
```

It is better to replace the previous example code with this sequence:

```
mul   r0, r1, r2
add   r3, r3, #1
sub   r4, r4, #1
sub   r5, r5, #1
cmp   r0, #0
```

Refer to Section 4.8, “Instruction Latencies for Intel XScale® Microarchitecture” for more information on instruction latencies for various multiply instructions. Schedule the multiply instructions taking into consideration their respective instruction latencies.

4.3.1.8 Scheduling SWP and SWPB Instructions

The SWP and SWPB instructions have a five-cycle issue latency. As a result of this latency, the instruction following the SWP/SWPB instruction stalls for 4 cycles. Only use the SWP/SWPB instructions where they are needed. For example, use SWP/SWPB to execute an atomic swap for a semaphore.

For example, the following code can be used to swap the contents of two memory locations. This code takes nine cycles to complete:

```
; Swap the contents of memory locations pointed to by r0 and r1
ldr   r2, [r0]
swp   r2, [r1]
str   r2, [r1]
```

This code takes six cycles to execute:

```
; Swap the contents of memory locations pointed to by r0 and r1
ldr   r2, [r0]
ldr   r3, [r1]
```

```
str   r2, [r1]
str   r3, [r0]
```

4.3.1.9 Scheduling the MRA and MAR Instructions (MRRC/MCRR)

The MRA (MRRC) instruction has an issue latency of one cycle, a result latency of two or three cycles depending on the destination register value being accessed, and a resource latency of two cycles. The code in the following example incurs a one-cycle stall due to the two-cycle resource latency of an MRA instruction.

```
mra   r6, r7, acc0
mra   r8, r9, acc0
add   r1, r1, #1
```

Rearrange the code to prevent the stall:

```
mra   r6, r7, acc0
add   r1, r1, #1
mra   r8, r9, acc0
```

Similarly, the following code incurs a two-cycle penalty due to the three-cycle result latency for the second destination register.

```
mra   r6, r7, acc0
mov   r1, r7
mov   r0, r6
add   r2, r2, #1
```

Rearrange the code to prevent the stall:

```
mra   r6, r7, acc0
add   r2, r2, #1
mov   r0, r6
mov   r1, r7
```

The MAR (MCRR) instruction has an issue latency, a result latency, and a resource latency of two cycles. Due to the two-cycle issue latency in this example, the pipeline always stalls for one cycle following an MAR instruction. Only use the MAR instruction when necessary.

4.3.1.10 Scheduling MRS and MSR Instructions

The issue latency of the MRS instruction is one cycle and the result latency is two cycles. The issue latency of the MSR instruction is two cycles (six if updating the mode bits) and the result latency is one cycle. The ORR instruction in the following example incurs a one cycle stall due to the two-cycle result latency of the MRS instruction.

```
mrs   r0, cpsr
orr   r0, r0, #1
add   r1, r2, r3
```

Move the ADD instruction to after the ORR instruction to prevent this stall.

4.3.1.11 Scheduling Coprocessor 15 Instructions

The MRC instruction has an issue latency of one cycle and a result latency of three cycles. The MCR instruction has an issue latency of one cycle. The MOV instruction in the following example, incurs a two-cycle latency due to the three-cycle result latency of the MRC instruction.

```
add   r1, r2, r3
mrc   p15, 0, r7, C1, C0, 0
mov   r0, r7
add   r1, r1, #1
```

Rearrange the code to avoid these stalls:

```
mrc   p15, 0, r7, C1, C0, 0
add   r1, r2, r3
add   r1, r1, #1
mov   r0, r7
```

4.3.2 Instruction Scheduling for Intel® Wireless MMX™ Technology

The Intel® Wireless MMX™ Technology provides an instruction set which offers the same functionality as the Intel® Wireless MMX™ Technology and Streaming SIMD Extensions (SSE) integer instructions.

4.3.2.1 Increasing Load Throughput on Intel® Wireless MMX™ Technology

The constraints on issuing load transactions with Intel XScale® Microarchitecture also hold with Intel® Wireless MMX™ Technology. The considerations reviewed using the Intel XScale® Microarchitecture instructions are re-illustrated in this section using the Intel® Wireless MMX™ Technology instruction set. The primary observations with load transactions are:

- The buffering in the memory pipeline allows two load-double transactions to be outstanding without incurring a penalty (stall).
- Back-to-back WLDRD instructions incur a stall, back-to-back WLDR(BHW) instructions do not incur a stall
- The WLDRD requires 4 cycles to return the DWORD assuming a cache hit, back-to-back WLDR (BHW) require 3 cycles to return the data.
- Use prefetching schemes with the above suggestions.

The overhead on issuing load transactions can be minimized by instruction scheduling and load pipelining. In most cases, it is straightforward to interleave other operation to avoid the penalty with back-to-back LDRD instructions. In the following code sequence, three WLDRD instructions are issued back to back incurring a stall on the second and third instruction.

```
WLDRD wR3,R4,#8
WLDRD wR5,r4,#8 - STALL
```

```
WLDRD wR4,R4,#8 -STALL
WADDB wR0,wR1,wR2
WADDB wR0,wR0,wR6
WADDB wR0,wR0,wR7
```

The same code sequence is reorganized to avoid a back-to-back issue of WLDRD instructions:

```
WLDRD wR3,R4,#8
WADDB wR0,wR1,wR2
WLDRD wR4,R4,#8
WADDB wR0,wR0,wR6
WLDRD wR5,r4,#8
WADDB wR0,wR0,wR7
```

Always try to separate 3 consecutive WLDRD instructions so that only 2 are outstanding at any one time and the loads are always interleaved with other instructions:

```
WLDRD wR0, [r2] , #8
WZERO wR15
WLDRD wR1, [r4] , #8
SUBS r3, r3, #8
WLDRD wR3, [r4] , #8
```

Always try to interleave additional operations between the load instruction and the instruction that first uses the data.

```
WLDRD wR0, [r2] , #8
WZERO wR15
WLDRD wR1, [r4] , #8
SUBS r3, r3, #8
WLDRD wR3, [r4] , #8
SUBS r4, r4, #1
WMACS R15, wR1, wR0
```

4.3.2.2 Scheduling the WMAC Instructions

The issue latency of the WMAC instruction is one cycle and the result and resource latency is two cycles. The second WMAC instruction in the following example stalls for one cycle due to the two-cycle resource latency.

```
WMACS wR0, wR2, wR3
WMACS wR1, wR4, wR5
```

The WADD instruction in the following example stalls for one cycle due to the two-cycle result latency.

```
WMACS wR0, wR2, wR3
WADD wR1, wR0, wR2
```

It is often possible to interleave instructions and effectively overlap their execution with multi-cycle instructions that utilize the multiply pipeline. The two-cycle WMAC instruction may be easily interleaved with operations that do not use the same resources:

```
WMACS wR14, wR2, wR3
WLDRD wR3, [r4] , #8
WMACS R15, wR1, wR0
WALIGNI wR5, wR6, wR7, #4
WMACS wR15, wR5, wR0
WLDRD wR0, [r3], #8
```

In the above example, the WLDRD and WALIGNI instructions do not incur a stall since they are using the memory and execution pipelines, respectively, and there are no data dependencies.

When using both Intel XScale® Microarchitecture and Intel® Wireless MMX™ Technology execution resources, it is also possible to overlap the multicycle instructions. The ADD instruction in the following example executes with no stalls:

```
WMACS wR14, wR1, wR2
ADD   R1, R2, R3
```

Refer to Section 4.8, “Instruction Latencies for Intel XScale® Microarchitecture” for more information on instruction latencies for various multiply instructions. Schedule the multiply instructions taking into consideration their respective instruction latencies.

4.3.2.3 Scheduling the TMIA Instruction

The issue latency of the TMIA instruction is one cycle and the result and resource latency are two cycles. The second TMIA instruction in the following example stalls for one cycle due to the two-cycle resource latency.

```
TMIA wR0, r2, r3
TMIA wR1, r4, r5
```

The WADD instruction in the following example stalls for one cycle due to the two-cycle result latency.

```
TMIA wR0, r2, r3
WADD wR1, wR0, wR2
```

Refer to Section 4.8, “Instruction Latencies for Intel XScale® Microarchitecture” for more information on instruction latencies for various multiply instructions. Schedule the multiply instructions taking into consideration their respective instruction latencies

4.3.2.4 Scheduling the WMUL and WMADD Instructions

The issue latency of the WMUL and WMADD instructions is one cycle and the result and resource latency are two cycles. The second WMUL instruction in the following example stalls for one cycle due to the two cycle resource latency.

```
WMUL wR0, wR1, wR2
WMUL wR3, wR4, wR5
```

The WADD instruction in the following example stalls for one cycle due to the two cycle result latency.

```
WMUL wR0, wR1, wR2
WADD wR1, wR0, wR2
```

4.4 SIMD Optimization Techniques

The Single Instruction Multiple Data (SIMD) architectures provided by the Intel® Wireless MMX™ Technology enables us to exploit the inherent parallelism found in the wide domain of multimedia and communication applications. The most time-consuming code sequences have certain characteristics in common:

- Operations are performed on small-native-data types (8-bit pixels, 16-bit voice, 32-bit audio)
- Regular and recurring memory-access patterns, usually data independent
- Localized, recurring computations performed on the data
- Compute-intensive processing

The following sections illustrate how the rules for writing fast sequences of Intel® MMX™ Technology instructions on Intel® Wireless MMX™ Technology can be applied to the optimization of short loops of Intel® MMX™ Technology code.

4.4.1 Software Pipelining

Software pipelining or loop unrolling is a well-known optimization technique where multiple calculations are in executed with each loop iteration. The disadvantages of applying this technique include: increases in code size for critical loops and restrictions on the minimum and multiples of taps or samples

The obvious advantage is in reduced cycle consumption. Overhead from loop-exit testing may be reduced, load-use stalls may be minimized and in some cases eliminated completely. Instruction scheduling opportunities may be created and exploited.

To illustrate the need for software pipelining, consider a key kernel of Intel® MMX™ Technology code that is central to many signal-processing algorithms: the real block Finite-Impulse-Response (FIR) filter. A real block FIR filter operates on two real vectors ***c(i)*** and ***x(i)*** and produces and output vector ***y(n)***. The vectors are represented for Intel® MMX™ Technology programming as arrays of 16-bit integers of some length N. The real FIR filter is represented by the equation:

$$y(n) = \sum_{i=0}^{L-1} c_i \cdot x(n-i), \qquad \vee\, 0 \le n \le N-1$$

or, in C-code,

```
for (i = 0; i < N; i++) {
   s = 0;
   for (j = 0; j < T; j++) {
   s += a[j]*x[i-j]);
   }
   y[i] = round (s);
}
```

The WMAC instruction is used for this calculation and provides for four parallel 16-bit-by-16-bit multiplications with accumulation. The first level of unrolling is a direct function of the four-way SIMD instruction that is used to implement the filter.

The C-code for the real block FIR filter is rewritten to illustrate that four taps are computed for each loop iteration.

```
for (i = 0; i < N; i++) {
   s0= 0;
   for (j = 0; j < T/4; j++4) {
      s0 += a[j]*x[i+j];
      s0 += a[j+1]*x[i+j+1];
      s0 += a[j+2]*x[i+j+2];
      s0 += a[j+3]*x[i+j+3];
   }
   y[i] = round (s0);
}
```

The direct assembly-code implementation of the inner loop illustrates clearly that optimum execution has not been accomplished. In the following code sequence, several undesirable stalls are present. The back-to-back LDRD instructions incur a one-cycle stall; the load-to-use penalty incurs a three-cycle stall. In addition, the loop overhead is high with two cycles being consumed for every four taps.

```
; Pointers r0 -> val , r1 -> pResult, r2 -> pTapsQ15 r3 -> tapsLen

    WZERO wR15
Loop_Begin:
    WLDRD wR0, [r2], #8
    WLDRD wR1, [r4], #8
```

```
   WMACS wR2, wR1, wR0
   SUBS r3, r3, #4
   BNE Loop_Begin
```

The parallelism of the filter may be exposed further by unrolling the loop to provide for eight taps per iteration. In the following code sequence, the loop has been unrolled once, eliminating several load-to-use stalls. The loop overhead has also been further amortized reducing it from two cycles for every four taps to two cycles for every eight taps. There is still a single load-to-use stall present between the second WLDRD instruction and the second WMACS instruction within the inner loop

```
; Pointers r0 -> val , r1 -> pResult, r2 -> pTapsQ15 r3 -> tapsLen

   WLDRD wR0, [r2] , #8
   WZERO wR15
   WLDRD wR1, [r4] , #8
Loop_Begin:
   WLDRD wR2, [r2] , #8
   SUBS r3, r3, #8
   WLDRD wR3, [r4] , #8
   WMACS wR15, wR1, wR0
   WLDRDNE wR0, [r2] , #8
   WMACS wR15, wR2, wR3
   WLDRDNE wR1, [r4] , #8
   BNE Loop_Begin
```

4.4.1.1 General Remarks on Software Pipelining

In the example for the real-block FIR filter, two copies of the basic sequence of code were interleaved, eliminating all but one of the stalls. The throughput for the sequence went from 9 cycles for every four taps to 9 cycles for every eight taps. This corresponds to a throughput of 1.125 cycles per tap represents a 2X throughput improvement.

It is useful to define a metric to describe the number of copies of a basic sequence of instructions that must be interleaved to remove all stalls. We can call this the interleave factor, k. The real block FIR filter requires k=2 to eliminate all possible stalls primarily because it is a small sequence that must take into account the long load-to-use latency. In practice, k=2 is sufficient for most loops encountered in real applications. This is fortunate because each interleaving requires its own set of temporary registers and with some algorithms, interleaving with k=3 is not possible. A good rule of thumb is to try k=2 first as it is usually the right choice.

4.4.2 Multi-Sample Technique

The multi-sample optimization technique provides for calculating multiple outputs with each loop iteration similar to loop unrolling. The disadvantages of applying this technique include increases in code size for critical loops. Restrictions on the minimum and multiples of taps or samples are also imposed. The obvious advantage is in reduced cycle consumption.

- Memory bandwidth is reduced by data re-use.
- Load-to-use stalls may be easily eliminated with scheduling.

- Multi-cycles may be interleaved with other instructions.

The C-code for the N-Sample, T-Tap block FIR filter is also used to illustrate the multi-sample technique.

```
for (i = 0; i < N; i++4) {
   s0=s1=s2=s3=0;
   for (j = 0; j < T/4; j++4) {
      s0 += a[j]*x[i-j];
      s1 += a[j]*x[i-j+1];
      s2 += a[j]*x[i-j+2];
      s3 += a[j]*x[i-j+3]);
   }
   y[i] = round (s0);
   y[i+1] = round (s1);
   y[i+2] = round (s2);
   y[i+3] = round (s3);
}
```

In the inner loop, we are calculating four output samples using the adjacent data samples *x(n-i)*, *x(n-1+1)*, *x(n-i+2)* and *x(n-i+3)*. The output samples *y(n)*, *y(n+1)*, *y(n+2)*, and *y(n+3)* are assigned to four 64-bit Intel® Wireless MMX™ Technology registers. To obtain near-ideal throughput, the inner loop is unrolled to provide for eight taps for each of the four output samples per loops iteration.

```
    ; ** Update pointers,
Outer_Loop:
    ; ** Update pointers,zero accumulators and prime the loop with DWORD loads

    WLDRD    wR0, [R1], #8    ; Load first 4 input samples
    WZERO    wR15
    WLDRD    wR1, [R1], #8    ; Load even groups of 4
                              ; input samples
    WZERO    wR14
    WLDRD    wR8, [R2], #8; Load first 4 coefficients
    WZERO    wR13
    WZERO    wR12

InnerLoop:
    ; ** Executes 8-Taps for each four outputs samples
    ; y(n),y(n+1), y(n+2),y(n+3)

    SUBS     R0  ,R0  , #8       ; Decrement loop counter
    WMAC     wR15,wR8 , wR0      ; y(n)+=
    WALIGNI  wR3 ,wR1 , wR0, #2
    WMAC     wR14,wR8 , wR3      ; y(n+1) +=
    WALIGNI  wR3 ,wR1 , wR0, #4
    WMAC     wR13,wR8 , wR3      ; y(n+2) +=
    WLDRD    wR0, [R1], #8       ;next 4 input samples
    WALIGNI  wR3 ,wR1 , wR0, #6
    WLDRD    wR9, [R2], #8       ; odd groups of 4 coeff.
    WMAC     wR12,wR8 , wR3      ; y(n+3) +=
```

```
WALIGNI  wR3 ,wR0 , wR1, #2
WALIGNI  wR4 ,wR0 , wR1, #4
WMAC     wR15,wR9 , wR1      ; y(n) +=
WALIGNI  wR5 ,wR0 , wR1, #6
WMAC     wR14,wR9 , wR3      ; y(n+1) +=
WLDRD    wR1, [R1], #8       ; even groups of 4 inputs
WMAC     wR13,wR9 , wR4      ; y(n+2) +=
WLDRD    wR8, [R2], #8       ; even groups of 4 coeff.
WMAC     wR12,wR8 , wR5      ; y(n+3) +=
BNE      Inner_Loop

; ** Outer loop code calculates the last four taps for
; y(n), y(n+1), y(n+2), y(n+3)**
; ** Store results

BNE Outer_Loop
```

4.4.2.1 General Remarks on Multi-Sample Technique

In the example for the real-block FIR filter, four outputs are computed simultaneously in the same inner loop, which has allowed the re-use of coefficients and sample data loaded into the register for computation of the first output to be used for the computation of the next three outputs. The interleave factor is set at k=2, which results in the elimination of load-to-use stalls. The throughput for the sequence is 20 cycles for every 32 taps, or 0.625 cycles per tap. This represents near ideal saturation of the execution resources.

The multi-sample technique may be applied whenever the same data is being used for multiple calculations. The large register file on Intel® Wireless MMX™ Technology facilitates this approach and a number of variations are possible.

4.4.3 Data Alignment Techniques

The exploitation of the data parallelism present in multimedia algorithms is accomplished by executing the same operation on different elements in parallel by packing several data elements into a single register and using the packed data instructions provided by the Intel® Wireless MMX™ Technology.

An important guideline for achieving optimum performance is always to align memory references, where an N-byte memory read or write is always on an N-byte boundary. In some cases, it is easy to align data so that all of the reads and writes are aligned. In others, it is more difficult because an algorithm naturally reads data in a misaligned fashion. A couple of examples include the single-sample FIR and video-motion estimation.

The Intel® Wireless MMX™ Technology provides a method for reducing the overhead associated with the classes of algorithms that require data to be accessed on 32-bit, 16-bit, or 8-bit binaries. The ALIGNI instruction is useful when the sequence of alignment is known beforehand as with the single-sample FIR filter. The ALIGNR instruction is useful when sequence of alignments are calculated when the algorithm executes as with the fast-motion search algorithms used in video compression. Both of these instructions operate on register pairs that may be effectively ping-ponged with alternate loads, significantly reducing the alignments overhead.

The following code sequence illustrates the set-up process for an unaligned array access. The procedure involves loading one of the general-purpose registers on Intel® Wireless MMX™ Technology with the lower three bits of the address pointer and then clearing the LSBs so that future accesses to the array are on a 64-bit boundary.

```
;r0 -> pointer to misaligned array.
MOV r5,#7
AND r7,r0,r5
TMCR wCGR1, r7
SUB r0,r0,r7;r0 psrc 64 bit aligned
```

Following the initial setup for alignment, the data can now be accessed, aligned, and presented to the execution resources.

```
WLDRD wR0, [r0]
WLDRD wR1, [r0,#8]
WALIGNR1 wR4, wR0, wR1
```

In the above code sequence, it is also necessary to interleave additional operations to avoid the back-to-back WLDRD and load-to-use penalties.

4.5 Porting Existing Intel® MMX™ Technology Code to Intel® Wireless MMX™ Technology

The re-use of existing Intel® MMX™ Technology code is encouraged since algorithm mapping to Intel® Wireless MMX™ Technology may be accelerated significantly. The Intel® MMX™ Technology target pipeline and architecture is different than Intel® Wireless MMX™ Technology and several changes are required for optimal mapping. The algorithms may require some redesign and attention to several aspects to make the task more manageable

- Data width – Intel® MMX™ Technology uses different designators for data types:
  - — Packed words for 16-bit operands, Intel® Wireless MMX™ Technology uses halfword (H)
  - — Packed double words for 32-bit operands, Intel® Wireless MMX™ Technology uses word (W)
  - — Quadwords for 64-bit operands, Intel® Wireless MMX™ Technology used doubleword (D)
- Instruction latencies – Instruction latencies are different with Intel® Wireless MMX™ Technology. May need to alter the scheduling of instructions.
- Instruction pairing – Intel® MMX™ Technology interleaves with x86 to reduce stalls. May need to alter the pairing of instructions in some cases on Intel® Wireless MMX™ Technology.
- Operand alignment – DWORD load/store requires 64-bit alignment. The pointers must be on a 64b boundary to avoid an exception.
- Memory latency – Memory latency for the PXA27x processor is different than existing Intel® MMX™ Technology.

Software pipelining, load pipelining and register rotation influences performance on the PXA27x processor.

- Increased register space - Intel® Wireless MMX™ Technology offers 16-doubleword registers.

  These registers can be used to store intermediate results and coefficients for tight multimedia inner loops without having to perform memory operations.

- The Intel® Wireless MMX™ Technology instructions provide encoding for three registers unlike the Intel® MMX™ Technology instructions, which provide for two registers only. The destination registers may be different from the source registers when converting Intel® MMX™ Technology code to Intel® Wireless MMX™ Technology. Remove all code sequences in Intel® MMX™ Technology that have MOV instructions associated with the destructive register behavior to improve throughput.

The following is an example of Intel® MMX™ Technology to Intel® Wireless MMX™ Technology instruction mapping.

Intel® Wireless MMX™ Technology Instructions	Intel® MMX™ Technology **Instructions**
WADDHSS wR0, wR0, wR7	PADDWSS mm0, mm7
WSUBHSS wR7, wR8, wR7	PSUBWSS mm7, mm8

4.5.1 Intel® Wireless MMX™ Technology Instruction Mapping

The following table shows the mapping of PXA27x processor instructions to Intel® Wireless MMX™ Technology and SSE integer instructions:

Table 4-1. PXA27x processor Mapping to Intel® Wireless MMX™ Technology and SSE (Sheet 1 of 2)

PXA27x processor	Intel® Wireless MMX™ Technology	SSE	Comments
WADD{b/h/w}	PADD{b/w/d}	—	
WSUB{b/h/w}	PSUB{b/w/d}	—	
WCMPEQ{b/h/w}	PCMPEQ{b/w/d}	—	
WCMPGT{b/h/w}	PCMPGT{b/w/d}	—	
WMUL{L}	PMULLW	—	
WMUL{H}	PMULHW	—	
WMADD	PMADDWD	—	
WSRA{h/w}	PSRA{w/d}	—	
WSLL{h/w/d}	PSLL{w/d/q}	—	
WSRL{h/w/d}	PSRL{w/d/q}	—	
WUNPCKIL{b/h/w}	PUNPCKL{bw/wd/dq}		PXA27x processor is a superset
WUNPCKIH{b/h/w}	PUNPCKH{bw/wd/dq}		PXA27x processor is a superset
WPACK{h/w}{SS}	PACKSS{wb/dw}	—	

Table 4-1. PXA27x processor Mapping to Intel® Wireless MMX™ Technology and SSE (Sheet 2 of 2)

PXA27x processor	Intel® Wireless MMX™ Technology	SSE	Comments
WPACK{h/w}{US}	PACKUS{wb/dw}	—	
WAND	PAND	—	
WANDN	PANDN	—	
WOR	POR	—	
WXOR	PXOR	—	
WMOV/WLDR	MOV{d/q}	—	
WMAX{B}{U}	—	PMAXUB	PXA27x processor is a superset
WMAX{H}{S}	—	PMAXSW	PXA27x processor is a superset
WMIN{B}{U}	—	PMINUB	PXA27x processor is a superset
WMAX{H}{S}	—	PMIXSW	PXA27x processor is a superset
TMOVMSK{B}	—	PMOVMSKB	Transfer instruction
WAVG2{B,W}	—	PAVG{BW}	PXA27x processor is a superset
TINSR{W}	—	PINSRW	PXA27x processor is a superset
TEXTRM{W}	—	PEXTRW	PXA27x processor is a superset
WSAD{B,W}	—	PSAD{BW}	
WSHUFH	—	PSHUFW	

Following is a set of examples showing subtle differences between Intel® MMX™ Technology and Intel® Wireless MMX™ Technology codes. The number of cases have been limited for the sake of brevity.

4.5.2 Unsigned Unpack Example

The Intel® Wireless MMX™ Technology provides instructions for unpacking 8-bit, 16-bit, or 32-bit data and either sign extending or zero extending.

The unsigned unpack replaces the Intel® MMX™ Technology sequence:

Intel® Wireless MMX™ Technology Instructions

Input: wR0 : Source Value

```
WUNPCKELU   wR1 ,  wR0
WUNPCKEHU   wR2 ,  wR0
```

Intel® MMX™ Technology Instructions

Input: mm0 : Source Value
mm7 : 0

```
MOVQ        mm1, mm0
PUNPCKLWD   mm0, mm7
PUNPCKHWD   mm1, mm
```

4.5.3 Signed Unpack Example

The signed unpack replaces the Intel® MMX™ Technology sequence:

Intel® Wireless MMX™ Technology Instructions			Intel® MMX™ Technology **Instructions**	
Input: wR0	: Source Value		Input: mm0	: Source Value
WUNPCKELS	wR1 ,	wR0	PUNPCKHWD	mm1, mm0
WUNPCKEHS	wR2 ,	wR0	PUNPCKLWD	mm0, mm0
			PSRAD	mm0, 16
			PSRAD	mm1, 16

4.5.4 Interleaved Pack with Saturation Example

This example uses signed words as source operands and the result is interleaved signed halfwords.

Intel® Wireless MMX™ Technology Instructions			Intel® MMX™ Technology **Instructions**	
Input: wR0	: Source Value 1		Input: mm0	: Source Value 1
wR1	: Source Value 2		mm1	: Source Value 2
WPACKWSS	wR2 ,	wR0, wR1	PACKSSDW	mm0, mm0
WSHUFH	wR2 ,	wR2, #216	PACKSSDW	mm1, mm1
			PUNPKLWD	mm0, mm1

4.6 Optimizing Libraries for System Performance

Many of the standard C library routines can benefit greatly by being optimized for the Intel XScale® Microarchitecture. The following string and memory manipulation routines are good candidates to be tuned for the Intel XScale® Microarchitecture.

strcat, strchr, strcmp, strcoll, strcpy, strcspn, strlen, strncat, strncmp, strpbrk, strrchr, strspn, strstr, strtok, strxfrm, memchr, memcmp, memcpy, memmove, memset

Apart from the C libraries, there are many critical functions that can be optimized in the same fashion. For example, graphics drivers and graphics applications frequently use a set of key functions. These functions can be optimized for the PXA27x processor. The following sections provide a set of routines as optimization case studies.

4.6.1 Case Study 1: Memory-to-Memory Copy

The performance of memory copy (memcpy) is influenced by memory-access latency and memory throughput. During memcpy, if the source and destination are both in cache, the performance is the highest and simple load-instruction scheduling can ensure the most efficient performance. However, if the source or the destination is not in the cache, a load-latency-hiding technique has to be applied.

Using preloads appropriately, the code can be desensitized to the memory latency (preload and prefetches are the same). Preloads are described further in Section 5.1.1.1.2, "Preload Loop Scheduling" on page 5-2. The following code performs memcpy with optimizations for latency desensitization.

```
        ; for cache-line-aligned case
        PLD [r5]
        PLD [r5, #32]
        PLD [r5, #64]
        PLD [r5, #96]

    LOOP :
        ldrd r0, [r5], #8
        ldrd r2, [r5], #8
        ldrd r4, [r5], #8
        str r0, [r6], #4
        str r1, [r6], #4
        ldrd r0, [r5], #8
        pld r5, #96]; preloading 3 cache lines ahead
        str r2, [r6], #4
        str r3, [r6], #4
        str r4, [r6], #4
        str r5, [r6], #4
        str r0, [r6], #4
        str r1, [r6], #4
....
```

This code preloads three cache lines ahead of its current iteration. It also uses **LDRD** and groups the **STR**s together to coalesce.

4.6.2 Case Study 2: Optimizing Memory Fill

Graphics applications use fill routines. Most of the personal-data assistant (PDA) LCD displays use output color format of RGB (16-bits or 8-bits). Therefore, most of the fill routines write out pixels as bytes or half-words, which is not recommended in terms of bus-bandwidth usage. However, multiple pixels can be packed into a 32-bit data format and used for writing to the memory. Use packing to improve efficiency.

Fill routines effectively make use of the write-coalescing feature which the PXA27x processor provides if the LCD frame buffer is allocated as un-cached but bufferable. This code example shows a common fill function:

```
unsigned short wColor, *pDst, DstStride;
BlitFill( ){
    for (int i = 0; i < iRows; i++) {
    // Set this solid color for whole scanline, then advance to next
        for (int j=0; j<iCols; j++)
            *pDst++ = wColor;
        pDst += DstStride;
    }
    return;
}
```

The optimized assembly is shown as:

```
        Start
        ; Set up code …

        ; Get wColor for each pixel arranged into hi and lo half-words
        ; of register so that multiple pixels can be written out
        orr r4,r1,r1,LSL #16

        ; code to check alignment of source and destination
        ; and code to handle end cases
        ; setup counters etc. is not shown
        ; Optimized loop may look like …
    LOOP
        str r4,[r0],#4 ; inner loop that fills destination scan line
        str r4,[r0],#4 ; pointed by r0
        str r4,[r0],#4 ; these stores take advantage of write coalescing
        str r4,[r0],#4
        str r4,[r0],#4 ; writing out as words
        str r4,[r0],#4 ; instead of bytes or half-words
        str r4,[r0],#4 ; achieves optimum performance
        str r4,[r0],#4

        str r4,[r0],#4
        str r4,[r0],#4
        str r4,[r0],#4
        str r4,[r0],#4

        str r4,[r0],#4
        str r4,[r0],#4
        str r4,[r0],#4
        str r4,[r0],#4

        subs r5,r5,#1 ;Fill 32 units(16 bits WORD) in each loop here
        bne LOOP
```

If the data is going to the internal memory, the same code offers even a greater throughput.

4.6.3 Case Study 3: Dot Product

Dot product is a typical vector operation for signal processing applications and graphics. For example, vertex transformation uses a graphic dot product. Using Intel® Wireless MMX™ Technology features can help accelerate these applications. The following code demonstrates how to attain this acceleration. These items are key issues for optimizing the dot-product code:

- Use LDRD if input is aligned
- Use the 2-cycle WMAC instruction
- Schedule around the load-to-use-latency

This example code is for the dot product:

```
        ; r0 points to source vector 1
        ; r1 points to source vector 2

        WLDRD wR0, [r0], #8
        WZERO wR15
        WLDRD wR1, [r1], #8
        MOV r10, #Eighth_Vector_len

Loop:
        WLDRD wR2, [r0], #8
        SUBS r10, #1
        WLDRD wR3, [r1], #8
        WMACS wR15, wR0, wR1
        WLDRDNE wR0, [r0], #8
        WMACS wR15, wR3, wR2
        WLDRDNE wR1, [r1], #8
        BNE Loop
```

4.6.4 Case Study 4: Graphics Object Rotation

Many handheld devices use native landscape orientation for internal graphics-application processing. However, if the end user views the output in portrait mode, a portrait-to-landscape conversion needs to occur each time the frame buffer writes to the display.

The display driver usually implements a landscape-to-portrait conversion when the frame is copied from the off-screen buffer to the display buffer. The following C code example shows a landscape to portrait rotation.

In the following example, *row* indicates the current row of the off-screen buffer. pDst and pSrc are single-byte pointers to the display and off-screen buffer, respectively.

```
for (row=Top; row < Bottom; row++) {
   for (j=0;j<2;j++) {
      *pDst++=*(pSrc+j);
   }
   pSrc-=bytesPerRow; // bytesPerRow = Stride
}
```

This is an optimized version of the previous example in assembly:

```
;Set up for loop goes here
;This shows only the critical loop in the implementation

LOOP:
ldr r5, [r0], r4            ; r0 = pSrc,
ldr r11, [r0], r4
ldr r8, [r0], r4
ldr r12, [r0], r4
```

```
; These loads are scheduled to distinct destination registers
and r6, r5, r9             ; r6->tmp = tmp0 & 0xffff;
orr r6, r6, r11, lsl #16   ; r6->tmp |= tmp1 << 16;
and r11, r11, r9,  lsl #16 ; r11->tmp1 &= 0xffff0000;
and r7, r8, r9             ; r7->tmp = tmp0 & 0xffff;
orr r11, r11, r5,  lsr #16 ; r11->tmp1 |= tmp0 >> 16;
orr r7, r7,  r12, lsl #16  ; r6->tmp |= tmp1 << 16;
str r6, [r1], #4           ; Write Coalesce the two stores
str r7, [r1], #4
and r12, r12, r9,  lsl #16 ; r11->tmp1 &= 0xffff0000;
orr r12, r12, r8,  lsr #16 ; r11->tmp1 |= tmp0 >> 16;
str r11, [r10], #4         ; Write Coalesce the two stores
str r12, [r10], #4
subs r14, r14, #1
bgt LOOP
```

In the following example, scheduled instructions take advantage of write-coalescing of multiple store instructions to the same line. In this example, the two stores are combined in a single write-buffer entry and issued as a single write request.

```
str r11, [r10], #4; Write Coalesce the two stores
str r12, [r10], #4
```

This can be exploited by either unrolling the C loop or by explicitly inlining multiple stores that can be combined.

The register rotation technique also allows multiple loads to be outstanding.

4.6.5 Case Study 5: 8x8 Block 1/2X Motion Compensation

Bi-linear interpolation is a typical operation in image and video processing applications. For example the video decode motion compensation uses the 1/2X interpolation operation. Using Intel® Wireless MMX™ Technology features can help to accelerate these key applications. The following code demonstrates how to attain this acceleration. These items are key issues for optimizing the 1/2X motion compensation:

- Use WALIGNR instruction for aligning the packed byte array.
- Use the WAVG2BR instruction for calculating the average of bytes.
- Schedule around the load-to-use-latency.

This example code is for the 1/2X interpolation:

```
; Test for special case of aligned ( LSBs = 110b and 000b)
; r0 -> pointer to misaligned array.
MOV r5,#7                  ; r5 =0x7
AND r7,r0,r5               ; r7 -> 3 LSBs of *psrc
MOV r12,#4                 ; counter
SUB r0,r0,r7               ; r0 psrc 64 bit aligned
WLDRD wR0, [r0]            ; load first group of 8 bytes
ADD r8,r7,#1               ; r8 -> 3LSBs of *psrc + 1
WLDRD wR1, [r0,#8]         ; load second group of 8 bytes
```

```
    TMCR wCGR1, r7              ; transfer alignment to wCGR1
    WLDRD wR2, [r0]
    TMCR wCGR2, r8
    WLDRD wR3, [r0,#8]

LOOP
    ; We recommend completely unrolling this loop it will save 8 cycles
    ADD r0,r0,r1
    SUBS r12,r12,#1
    WALIGNR1 wR4, wR0, wR1
    WALIGNR2 wR5, wR0, wR1
    WLDRD wR0, [r0]
    WAVG2BR wR6, wR4, wR5
    WLDRD wR1, [r0,#8]
    SUB r12,r12,#1
    WSTRD wR6, [r2]
    WALIGNR1 wR4, wR2, wR3
    WALIGNR2 wR5, wR2, wR3
    WLDRD wR2, [r0]
    WAVG2BR wR6, wR4, wR5
    WLDRD wR3, [r0,#8]
    ADD r2,r2,r3 ; Adding stride
    WSTRD wR6, [r2]
    ADD r2,r2,r3 ; Adding stride
    BNE LOOP
```

4.7 Intel® Performance Primitives

Users who want to take full advantage of many of the optimizations in this guide are likely to use these techniques:

- Write hand-optimized assembly code.
- Use a compiler tuned for the Intel XScale® Microarchitecture and the ARM* v5TE instruction set architecture.
- Incorporate a library with optimizations present.

For the last item, a listing of fully optimized code, the Intel® Integrated Performance Primitives (IPP) is available. The IPP comprises a rich and powerful set of general and multimedia signal-processing kernels optimized for high performance on the PXA27x processor. Besides optimization, the IPP offers application developers a number of significant advantages, including accelerated time to market, compatibility with many major real-time embedded operating systems, and support for porting across certain Intel® platforms.

The IPP include optimized general signal- and image-processing primitives, as well as primitives for use in constructing internationally standardized audio, video, image, and speech encoder/decoders (CODECs) for the PXA27x processor.

IPP available for general one-dimensional (1D) signal processing include:

- Vector initialization, arithmetic, statistics, thresholding, and measure
- Deterministic and random signal generation
- Convolution, filtering, windowing, and transforms

IPP for general two-dimensional (2D) image processing include:

- Vector initialization, arithmetic, statistics, thresholding, and measure
- Color conversions
- Morphological operations
- Convolution, filtering, windowing, and transforms

Additional IPP are available allowing construction of these multimedia CODECs:

- Video - ITU H.263 decoder, ISO/IEC 14496-2 MPEG-4 decoder
- Audio - ISO/IEC 11172-3 and 13818-3 (MPEG-1, -2) Layer 3 (“MP3”) decoder.
- Speech - ITU-T G.723.1 CODEC and ETSI GSM-AMR codec
- Image - ISO/IEC JPEG CODEC

For more details on the IPP, including libraries for 3D graphics and encryption, browse http://intel.com/software/products/ipp/.

4.8 Instruction Latencies for Intel XScale® Microarchitecture

The following sections show the latencies for all the instructions with respect to their functional groups: branch, data processing, multiply, status register access, load/store, semaphore, and coprocessor.

Section 4.8.1, “Performance Terms” explains how to read Table 4-2 through Table 4-17.

4.8.1 Performance Terms

- Issue Clock (cycle 0)

  The first cycle when an instruction is decoded and allowed to proceed to additional stages in the execution pipeline.

- Cycle Distance from A to B

  The cycle distance from cycle *A* to cycle *B* is (*B*-*A*) — the number of cycles from the start of cycle *A* to the start of cycle *B*. For example, the cycle distance from cycle 3 to cycle 4 is one cycle.

- Issue Latency

  The cycle distance *from* the first issue clock of the current instruction *to* the issue clock of the next instruction. Cache misses, resource-dependency stalls, and resource-availability conflicts can influence the actual number of cycles.

- Result Latency

The cycle distance *from* the first issue clock of the current instruction *to* the issue clock of the first instruction using the result without incurring a resource dependency stall. Cache misses, resource-dependency stalls, and resource-availability conflicts influence the actual number of cycles.

- Minimum Issue Latency (without branch misprediction)

  This represents the minimum cycle distance *from* the issue clock of the current instruction *to* the first possible issue clock of the next instruction. For example, the issuing of the next instruction is not stalled due to these situations:

  — Resource-dependency stalls

  — The next instruction can be fetched immediately from the cache or memory interface

  — The current instruction does not incur a resource-dependency stall during execution that can not be detected at its issue time

  — The instruction uses dynamic-branch prediction, correct prediction is assumed.

- Minimum Result Latency

  This represents the required minimum cycle, which is the distance *from* the issue clock of the current instruction *to* the issue clock of the first instruction that uses the result without incurring a resource dependency stall. For example, the issuing of the next instruction is not stalled due to these situations:

  — Resource-dependency stalls

  — The next instruction can be immediately fetched from the cache or memory interface.

  — The current instruction does not incur resource-dependency stalls during executions that cannot be detected at issue time.

- Minimum Issue Latency (with branch misprediction)

  It represents the minimum cycle distance *from* the issue clock of the current branching instruction *to* the first possible issue clock of the next instruction. The value of this is identical to the minimum-issue latency except the branching instruction is mispredicted. It is calculated by adding minimum issue latency (without branch misprediction) to the minimum branch latency penalty cycles using Table 4-3 and Table 4-4.

- Minimum Resource Latency

  The minimum cycle distance *from* the issue clock of the current multiply instruction *to* the issue clock of the next multiply instruction assuming the second multiply does not incur a data dependency and is immediately available from the instruction cache or memory interface.

  This code is an example of computing latencies:

```
UMLAL r6,r8,r0,r1
ADD r9,r10,r11
SUB r2,r8,r9
MOV r0,r1
```

Table 4-2 shows how to calculate issue latency and result latency for each instruction. The UMLAL instruction (shown in the issue column) starts to issue on cycle 0 and the next instruction, ADD, issues on cycle 2, so the issue latency for UMLAL is two. From the code fragment, there is a result dependency between the UMLAL instruction and the SUB instruction. In Table 4-2, UMLAL starts to issue at cycle 0 and the SUB issues at cycle 5, thus the result latency is five.

Table 4-2. Latency Example

Cycle	Issue	Executing
0	umlal (1st cycle)	—
1	umlal (2nd cycle)	umlal
2	add	umlal
3	sub (stalled)	umlal & add
4	sub (stalled)	umlal
5	sub	umlal
6	mov	sub
7	—	mov

4.8.2 Branch Instruction Timings

Table 4-3. Branch Instruction Timings (Those Predicted By the BTB (Branch Target Buffer))

Instruction	Minimum Issue Latency When Correctly Predicted By The Btb	Minimum Issue Latency With Branch Misprediction
B	1	5
BL	1	5

Table 4-4. Branch Instruction Timings (Those Not Predicted By the BTB)

Instruction	Minimum Issue Latency When the Branch Is Not Taken	Minimum Issue Latency When the Branch is Taken
BLX(1)	—	5
BLX(2)	1	5
BX	1	5
Data Processing Instruction with PC as the destination	Same as Table 4-5	4 + numbers in Table 4-5
LDR PC,<>	2	8
LDM with PC in register list	3 + numreg†	10 + max (0, numreg-3)
† numreg is the number of registers in the register list including the PC.		

4.8.3 Data Processing Instruction Timings

Table 4-5. Data Processing Instruction Timings

Instruction	<shifter operand> Is Not a Shift/Rotate by Register		<shifter operand> is a Shift/Rotate by Register Or <shifter operand> is RRX	
	Minimum Issue Latency	Minimum Result Latency†	Minimum Issue Latency	Minimum Result Latency†
ADC	1	1	2	2
ADD	1	1	2	2
AND	1	1	2	2
BIC	1	1	2	2
CMN	1	1	2	2
CMP	1	1	2	2
EOR	1	1	2	2
MOV	1	1	2	2
MVN	1	1	2	2
ORR	1	1	2	2
RSB	1	1	2	2
RSC	1	1	2	2
SBC	1	1	2	2
SUB	1	1	2	2
TEQ	1	1	2	2
TST	1	1	2	2

† If the next instruction uses the result of the data processing for a shift by immediate or as Rn in a QDADD or QDSUB, one extra cycle of result latency is added to the number listed above

4.8.4 Multiply Instruction Timings

Table 4-6. Multiply Instruction Timings (Sheet 1 of 2)

Instruction	Rs Value (Early Termination)	S-Bit Value	Minimum Issue Latency	Minimum Result Latency†	Minimum Resource Latency (Throughput)
MLA	Rs[31:15] = 0x00000 or Rs[31:15] = 0x1FFFF	0	1	2	1
		1	2	2	2
	Rs[31:27] = 0x00 or Rs[31:27] = 0x1F	0	1	3	2
		1	3	3	3
	all others	0	1	4	3
		1	4	4	4
MUL	Rs[31:15] = 0x00000 or Rs[31:15] = 0x1FFFF	0	1	2	1
		1	2	2	2
	Rs[31:27] = 0x00 or Rs[31:27] = 0x1F	0	1	3	2
		1	3	3	3
	all others	0	1	4	3
		1	4	4	4
SMLAL	Rs[31:15] = 0x00000 or Rs[31:15] = 0x1FFFF	0	2	RdLo = 2; RdHi = 3	2
		1	3	3	3
	Rs[31:27] = 0x00 or Rs[31:27] = 0x1F	0	2	RdLo = 3; RdHi = 4	3
		1	4	4	4
	all others	0	2	RdLo = 4; RdHi = 5	4
		1	5	5	5
SMLALxy	N/A	N/A	2	RdLo = 2; RdHi = 3	2
SMLAWy	N/A	N/A	1	3	2
SMLAxy	N/A	N/A	1	2	1
SMULL	Rs[31:15] = 0x00000 or Rs[31:15] = 0x1FFFF	0	1	RdLo = 2; RdHi = 3	2
		1	3	3	3
	Rs[31:27] = 0x00 or Rs[31:27] = 0x1F	0	1	RdLo = 3; RdHi = 4	3
		1	4	4	4
	all others	0	1	RdLo = 4; RdHi = 5	4
		1	5	5	5
SMULWy	N/A	N/A	1	3	2
SMULxy	N/A	N/A	1	2	1
UMLAL	Rs[31:15] = 0x00000	0	2	RdLo = 2; RdHi = 3	2
		1	3	3	3
	Rs[31:27] = 0x00	0	2	RdLo = 3; RdHi = 4	3
		1	4	4	4
	all others	0	2	RdLo = 4; RdHi = 5	4
		1	5	5	5

Table 4-6. Multiply Instruction Timings (Sheet 2 of 2)

Instruction	Rs Value (Early Termination)	S-Bit Value	Minimum Issue Latency	Minimum Result Latency†	Minimum Resource Latency (Throughput)
UMULL	Rs[31:15] = 0x00000	0	1	RdLo = 2; RdHi = 3	2
		1	3	3	3
	Rs[31:27] = 0x00	0	1	RdLo = 3; RdHi = 4	3
		1	4	4	4
	all others	0	1	RdLo = 4; RdHi = 5	4
		1	5	5	5

† If the next instruction needs to use the result of the multiply for a shift by immediate or as Rn in a QDADD or QDSUB, one extra cycle of result latency is added to the number listed.

Table 4-7. Multiply Implicit Accumulate Instruction Timings

Instruction	Rs Value (Early Termination)	Minimum Issue Latency	Minimum Result Latency	Minimum Resource Latency (Throughput)
MIA	Rs[31:15] = 0x0000 or Rs[31:15] = 0xFFFF	1	1	1
	Rs[31:27] = 0x0 or Rs[31:27] = 0xF	1	2	2
	all others	1	3	3
MIAxy	N/A	1	1	1
MIAPH	N/A	1	2	2

Table 4-8. Implicit Accumulator Access Instruction Timings

Instruction	Minimum Issue Latency	Minimum Result Latency	Minimum Resource Latency (Throughput)
MAR	2	2	2
MRA	1	(RdLo = 2; RdHi = 3)†	2

† If the next instruction needs to use the result of the MRA for a shift by immediate or as Rn in a QDADD or QDSUB, one extra cycle of result latency is added to the number listed.

4.8.5 Saturated Arithmetic Instructions

Table 4-9. Saturated Data Processing Instruction Timings

Instruction	Minimum Issue Latency	Minimum Result Latency
QADD	1	2
QSUB	1	2
QDADD	1	2
QDSUB	1	2

4.8.6 Status Register Access Instructions

Table 4-10. Status Register Access Instruction Timings

Instruction	Minimum Issue Latency	Minimum Result Latency
MRS	1	2
MSR	2 (6 if updating mode bits)	1

4.8.7 Load/Store Instructions

Table 4-11. Load and Store Instruction Timings

Instruction	Minimum Issue Latency	Minimum Result Latency
LDR	1	3 for load data; 1 for writeback of base
LDRB	1	3 for load data; 1 for writeback of base
LDRBT	1	3 for load data; 1 for writeback of base
LDRD	1 (+1 if Rd is R12)	3 for Rd; 4 for Rd+1; 1 (+1 if Rd is R12) for writeback of base
LDRH	1	3 for load data; 1 for writeback of base
LDRSB	1	3 for load data; 1 for writeback of base
LDRSH	1	3 for load data; 1 for writeback of base
LDRT	1	3 for load data; 1 for writeback of base
PLD	1	N/A
STR	1	1 for writeback of base
STRB	1	1 for writeback of base
STRBT	1	1 for writeback of base
STRD	2	2 for writeback of base
STRH	1	1 for writeback of base
STRT	1	1 for writeback of base

Table 4-12. Load and Store Multiple Instruction Timings

Instruction	Minimum Issue Latency	Minimum Result Latency
LDM†	2 + numreg††	5-18 for load data (4 + numreg for last register in list; 3 + numreg for 2nd to last register in list; 2 + numreg for all other registers in list); 2+ numreg for writeback of base
STM	2 + numreg	2 + numreg for writeback of base

† See Table 4-4 for LDM timings when R15 is in the register list

†† numreg is the number of registers in the register list

4.8.8 Semaphore Instructions

Table 4-13. Semaphore Instruction Timings

Instruction	Minimum Issue Latency	Minimum Result Latency
SWP	5	5
SWPB	5	5

4.8.9 CP15 and CP14 Coprocessor Instructions

Table 4-14. CP15 Register Access Instruction Timings

Instruction	Minimum Issue Latency	Minimum Result Latency
MRC†	4	4
MCR	2	N/A

† MRC to R15 is unpredictable / MRC and MCR to CP0 and CP1 is described in the Intel® Wireless MMX™ Technology section

Table 4-15. CP14 Register Access Instruction Timings

Instruction	Minimum Issue Latency	Minimum Result Latency
MRC	8	8
MRC to R15	9	9
MCR	8	N/A
LDC	11	N/A
STC	8	N/A

4.8.10 Miscellaneous Instruction Timing

Table 4-16. Exception-Generating Instruction Timings

Instruction	Minimum latency to first instruction of exception handler
SWI	6
BKPT	6
UNDEFINED	6

Table 4-17. Count Leading Zeros Instruction Timings

Instruction	Minimum Issue Latency	Minimum Result Latency
CLZ	1	1

4.8.11 Thumb* Instructions

In general, the timing of THUMB* instructions is the same as their equivalent ARM* instructions, except for these cases:

- If the equivalent ARM* instruction maps to an entry in Table 4-3, the "Minimum Issue Latency with branch misprediction" goes from 5 to 6 cycles due to the branch latency penalty.
- If the equivalent ARM* instruction maps to one in Table 4-4, the "Minimum Issue Latency when the Branch is Taken" increases by one cycle due to the branch latency penalty.
- The timings of a THUMB* BL instruction and an ARM* data processing instruction when H=0 are the same.

The mapping of THUMB* instructions to ARM* instructions can be found in the *ARM*Architecture Reference Manual*

4.9 Instruction Latencies for Intel® Wireless MMX™ Technology

The issue cycle and result latency of all the PXA27x processor instructions is shown in Table 4-18. In this table, the issue cycle is the number of cycles that an instruction takes to leave the register file. The **result latency** is the number of cycles required to calculate the result and make it available to the bypassing logic. A result latency of *1* indicates that the value is available immediately to the following instruction. Table 4-18 shows the best case result latency that can be degraded by data or resource hazards.

Table 4-18. Issue Cycle and Result Latency of the PXA27x processor Instructions (Sheet 1 of 2)

Instructions	Issue Cycle	Result Latency
WADD	1	1
WSUB	1	1
WCMPEQ	1	2
WCMPGT	1	2
WAND	1	1
WANDN	1	1
WOR	1	1
WXOR	1	1
WAVG2	1	1
WMAX	1	2
WMIN	1	2
WSAD	1	1
WACC	1	1
WMUL	1	1
WMADD	1	1
WMAC	1	2

Table 4-18. Issue Cycle and Result Latency of the PXA27x processor Instructions (Sheet 2 of 2)

Instructions	Issue Cycle	Result Latency
TMIA	1	2
TMIAPH	1	1
TMIAxy	1	1
WSLL	1	1
WSRA	1	1
WSRL	1	1
WROR	1	1
WPACK	1	2
WUNPCKEH	1	1
WUNPCKEL	1	1
WUNPCKIH	1	1
WUNPCKIL	1	1
WALIGNI	1	1
WALIGNR	1	1
WSHUF	1	1
TANDC	1	1
TORC	1	1
TEXTRC	1	1
TEXTRM	1	2
TMCR	1	3
TMCRR	1	1
TMRC	1	2
TMRRC	1	3
TMOVMSK	1	2
TINSTR	1	1
TBCST	1	1
WLDR (BHW) to main regfile	1	4 (3)†, ††
WLDRW to control regfile	1	4††
WSTR	1	na††

† WLDRD is 4 cycles WLDR<B,H,W> is 3 cycles

†† Base address register update for WLDR and WSTR is the same as the core load/store operation

4.10 Performance Hazards

The basic performance of the system can be effected by stalls caused by data or resource hazards. This section describes the factors effecting each type of hazard and the implications for performance.

4.10.1 Data Hazards

A data hazard occurs when an instruction requires data that cannot be provided by the register file or the data-forwarding mechanism, or if two instructions update the same destination register in an out-of-order fashion. The first hazard is termed as *read-after-write (RAW)* and the second hazard is termed as *write-after-write (WAW)*. The processing of the new instruction is stalled until the data becomes available for RAW hazards, and until it can be guaranteed that the new instruction updates the register file after previous the instruction has updated the same destination register, for WAW hazards. The PXA27x processor device contains a bypassing mechanism for ensuring that data and different stages of the pipeline can forwarded to the correct instructions. There are, however, certain combinations of instructions where it is not possible to forward directly between instructions in the PXA27x processor 1.0 implementation.

The result latency shown in Table 4-18 and best-case result latency are generally achievable. However there are certain instruction combinations where these result latencies do not hold because not all combinations of bypassing logic exist in the hardware, and some instructions require more time to calculate the result when certain qualifiers are specified. This list describes the data hazards for the PXA27x processor 1.0 implementation:

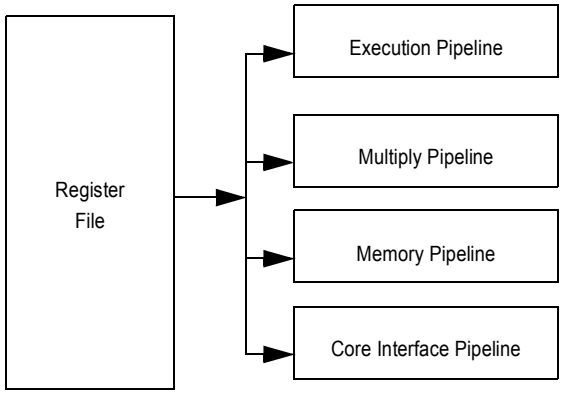
- When saturation is specified for WADD or WSUB, the result latency is increased to two cycles
- The destination register (accumulator) for certain multiplier instructions (WMAC, WSAD, TMIA, TMIAph, TMIAxy) can be forwarded for accumulation to the same destination register only. If the destination register results are needed by another instruction as source operands, there is an additional result latency as the result is available from the regular forwarding paths, external to the multiplier. The exact number of extra cycles depends on the multiplier instruction that is delivering results to source operands of other instructions.
- If an instruction is updating a destination register from the multiply pipeline, a following instruction in the execute, memory or core interface pipelines updating the same destination register is stalled until it can be guaranteed that the following instruction updates the register file after the previous instruction in the multiply pipe has updated the register file
- If an instruction is updating a destination register from the memory pipeline, a following instruction updating the same destination register is stalled until it can be guaranteed that the following instruction updates the register file after the previous instruction in the memory pipe has updated the register file.
- If the Intel XScale® Microarchitecture MAC unit is in use, the resulting latency of a TMRC, TMRRC, and TEXRM increases accordingly.

4.10.2 Resource Hazard

A **resource hazard** is caused when an instruction requires a resource that is already in use. When this condition is detected, the processing of the new instruction is stalled at the register file stage.

Figure 4-1 shows a high-level representation of the operation of the PXA27x processor coprocessor. After the register file, there are four concurrent pipelines to which an instruction can be dispatched. An instruction can be issued to a pipeline if the resource is available and there are no unresolved data dependencies. For example, a load instruction that uses the memory pipeline can be issued while a multiply instruction is completing in the multiply pipeline (assuming there are no data hazards.)

Figure 4-1. High-Level Pipeline Organization



The performance effect of resource contention can be quantified by examining the delay taken for a particular instruction to release the resource after starting execution. The definition of "release the resource" in this context is that the resource can accept another instruction (note: the resource may still be processing the previous instruction further down its internal pipeline). A delay of one clock cycle indicates that the resource is available immediately to the next instruction. A delay greater than one clock cycle stalls the next instruction if the same resource is required. The following sections examine the resource-usage delays for the four pipelines, and how these map onto the instruction set.

4.10.2.1 Execution Pipeline

An instruction can be accepted into the execution pipeline when the first stage of the pipeline is empty. Table 4-19 shows the instructions that execute in the main execution pipeline. All these instructions have a resource-use delay of one clock cycle. Therefore, the execution pipeline is always available to the next instruction.

Table 4-19. Resource Availability Delay for the Execution Pipeline (Sheet 1 of 2)

Instructions	Delay (Clocks)
WADD	1
WSUB	1
WCMPEQ	1
WCMPGT	1
WAND	1
WANDN	1
WOR	1
WXOR	1
WAVG2	1
WMAX	1
WMIN	1
WSAD	1†

Table 4-19. Resource Availability Delay for the Execution Pipeline (Sheet 2 of 2)

Instructions	Delay (Clocks)
WSLL	1
WSRA	1
WSRL	1
WROR	1
WPACK	1
WUNPCKEH	1
WUNPCKEL	1
WUNPCKIH	1
WUNPCKIL	1
WALIGNI	1
WALIGNR	1
WSHUF	1
TMIA	1†
TMIAph	1†
TMIAxy	1†
TMCR	1
TMCRR	1
TINSR	1
TBCST	1
TANDC	1
TORC	1
TEXTRC	1

† The WSAD, TMIA, TMIAph, TMIAxy execute in both the main execution pipeline and the multiplier pipeline. They execute for one cycle in the execution pipeline and the rest in the multiplier pipeline. See Section 4.10.2.5 for more details

4.10.2.2 Multiply Pipeline

Instructions issued to the multiply pipeline may take up to two cycles before another instruction can be issued to the pipeline. The instructions in the multiply pipe can be categorized into 4 classes shown in Table 4-20. The resource-availability delay for the instructions that are mapped onto the multiplier pipeline depend on the class of the multiply instruction that subsequently wants to use the multiply resource. These delays are shown in Table 4-21. For example if a TMIA instruction is followed by a TMIAph (class3) instruction, then the TMIAph sees a resource availability of 2 cycles.

Table 4-20. Multiply pipe instruction classes (Sheet 1 of 2)

Instructions	Class
WACC	1
WMAC, WMUL, WMADD	2

Table 4-20. Multiply pipe instruction classes (Sheet 2 of 2)

WSAD, TMIAph, TMIAxy	3
TMIA	4

Table 4-21. Resource Availability Delay for the Multiplier Pipeline

Instruc-tions	Delay(Clocks) for a subsequent class 1 multiply pipe instruction	Delay(Clocks) for a subsequent class 2 multiply pipe instruction	Delay(Clocks) for a subsequent class 3 multiply pipe instruction	Delay(Clocks) for a subsequent class 4 multiply pipe instruction
WSAD	2	2	1	1
WACC	1	1	1	1
WMUL	2	2	1	1
WMADD	2	2	1	1
WMAC	2	2	1	1
TMIA	3	3	2	2
TMIAPH	2	2	1	1
TMIAxy	2	2	1	1

WSAD, TMIA, TMIAxy, TMIAph execute in both the main execution pipeline and the multiplier pipeline. See Section 4.10.2.5 for more details

4.10.2.3 Memory Control Pipeline

The memory control pipeline is responsible for coordinating the load/store activity with the main core. The external interface to memory is 32-bits so the 64-bit load/store issued by the PXA27x processor device is sequenced as two 32-bit load/stores to memory. This is transparent to end users and is already factored into the result latencies show in Table 4-18. After the PXA27x processor device issues the 64-bit memory transaction, it must buffer the data until the two 32-bit half transactions are complete. Currently, there are two 64-bit buffer slots for load operations and one 64-bit buffer slot available for store transactions. If the memory buffer is currently empty, the Memory pipeline resource- availability delay is only one clock. However, if the buffer is currently full due to a sequence of memory transactions, the following instruction must wait for space in the buffer. The resource availability delay in this case is two cycles and is summarized in Table 4-22.

Table 4-22. Resource Availability Delay for the Memory Pipeline

Instructions	Delay(Clocks)	Condition
WLDRD	1	Two loads not already outstanding
WSTRD	2	
WLDRD	3+M	Two loads already outstanding (M is delay for main memory if cache miss)

4.10.2.4 Coprocessor Interface Pipeline

The coprocessor interface pipeline also contains buffering to allow multiple outstanding MRC/MRRC operations. The coprocessor interface pipeline can continue to accept MRC and MRRC instructions every cycle until its buffers are full. Currently there is sufficient storage in the buffer for either four MRC data values (32-bit) or two MRRC data values (64-bit).Table 4-23 shows a summary of the resource availability delay for the coprocessor interface.

Table 4-23. Resource Availability Delay for the Coprocessor Interface Pipeline

Instructions	Delay(Clocks)	Condition
TMRC	1	Buffer Empty
TMRC	2	Buffer Full
TMRRC	1	Buffer empty
TMRRC	2	Buffer Full

There is also an interaction between TMRC/TMRRC and any instructions in the core that use the MAC unit of the core. For optimum performance, do not use the MAC unit in the core adjacent to TMRC instructions as they both share the route back to the core register file.

4.10.2.5 Multiple Pipelines

The WSAD, TMIA, TMIAph and TMIAxy instructions execute in both the main execution pipeline and the multiplier pipeline. The instruction executes one cycle in the execution pipeline and the rest in the multiplier pipeline. The WSAD, TMIA, TMIAph, TMIAxy instructions always issue without stalls to the execution pipeline (see Section 4.10.2.1). The availability of the multiplier pipeline depends on a previous instruction that was using the multiply resource. If the previous instruction was a TMIA, there is an effective resource availability of two cycles.

§§

High Level Language Optimization 5

5.1 C and C++ Level Optimization

For embedded systems, system performance is greatly affected by software programming techniques. To attain performance at the application level, there are many techniques that can be applied at the C/C++ code development phase. This chapter covers a set of programming optimization techniques that are relevant to embedded systems such as phones and PDAs based on the Intel® PXA27x Processor Family (PXA27x processor).

5.1.1 Efficient Usage of Preloading

The Intel XScale® Microarchitecture preload instruction is a true preload instruction because the load destination is the data or mini-data cache and not a register. Compilers for processors that have data caches, but do not support preload, sometimes use a load instruction to preload the data cache. This technique has the disadvantage of using a register to load data and requiring additional registers for subsequent preloads, thus increasing register pressure. By contrast, the Intel XScale® Microarchitecture preload can be used to reduce register pressure instead of increasing it.

The Intel XScale® Microarchitecture preload is a hint instruction and does not guarantee that the data is loaded. Whenever the load causes a fault or a table walk, the processor ignores the preload instruction, the fault or table walk, and continues processing the next instruction. This is particularly advantageous in the case where a linked list or recursive data structure is terminated by a NULL pointer. Preloading the NULL pointer does not cause a fault.

The preload instructions (PLD) can be inserted by the compiler during compilation. However, programmers can effectively insert preload operations in the code. A function can be defined during high-level language programming, which results in a PLD instruction being inserted in line. This function can be called at other suitable places in the code to insert PLD instructions.

5.1.1.1 Preload Considerations

The issues associated with using preloading that require consideration are explained below.

5.1.1.1.1 Preload Distances In the Intel XScale® Microarchitecture

Scheduling the preload instruction requires understanding the system latency times and system resources that determine when to use the preload instruction.

The optimum advantage of using preload is obtained if the preload issue-to-use distance is equal to the memory latency. Use the memory latency shown in Section 3.2.1, “Optimal Setting for Memory Latency and Bandwidth” to determine the proper insertion point for preloads.

The preload distance may need to be varied depending on whether the target is in the internal memory or in the external memory. For external memory in which the target address is not aligned to a cache line, the memory latency can increase due to the critical-word-first (CWF) mode of the memory accesses. CWF mode returns the requested data starting with the requested word instead of starting with the word at the aligned address.When using preloads, align the target address to a cache-line boundary to avoid the extra memory bus usage.

Consider this code sample:

```
   add   r1, r1, #1
; Sequence of instructions using r2, but leave r3 unchanged.

   ldr   r2, [r3]
   add   r3, r3, #4
   mov   r4, r3
   sub   r2, r2, #1
```

The sub instruction above would stall if the data being loaded misses the cache. These stalls can be avoided by using a PLD instruction as:

```
   pld   [r3]
   add   r1, r1, #1

; Sequence of instructions using r2, but leave r3 unchanged.
   ldr   r2, [r3]
   add   r3, r3, #4
   mov   r4, r3
   sub   r2, r2, #1
```

For most cases, optimizing for the external memory latency also satisfies the requirements for the internal memory latency.

5.1.1.1.2 Preload Loop Scheduling

When adding preload instructions to a loop that operates on arrays, preload ahead one, two, or more iterations. The data for future iterations is located in memory a fixed offset from the data for the current iteration, which makes it easy to predict where to fetch the data. The number of iterations to preload ahead is referred to as the *preload scheduling distance (PSD)*. For the Intel XScale® Microarchitecture this PSD can be calculated as:

$$PSD = floor\left(\frac{(N_{linexfer} \times N_{pref} + N_{hwlinexfer} \times N_{evict})}{(CPI \times N_{inst})}\right)$$

Where:

$N_{linexfer}$ The number of core clocks required to transfer one complete cache line.

N_{pref} The number of cache lines to be pre-loaded for both reading and writing.

N_{evict} The number of cache half line evictions caused by the loop.

N_{inst} The number of instructions executed in one iteration of the loop

$N_{hwlinexfer}$ The number of core clocks required to write half a cache line (as if) only one of the cache line dirty bits were set when a line eviction occurred.

CPI The average number of core clocks per instruction (of the instructions within the loop).

PSD calculated in the above equation is a good initial estimation, but may not be the optimum scheduling distance. Estimating N_{evict} is difficult from static code. However, if the operational data uses the mini-data cache and if the loop operations overflow the mini-data cache, then a first-order estimate of N_{evict} is the number of bytes written per loop iteration divided by a half-cache-line size

(16 bytes). Cache overflow can be estimated by the number of cache lines transferred each iteration and the number of expected loop iterations. N_{evict} and CPI can be estimated by profiling the code using the performance monitor “cache write-back” event count.

Consider CWF latency in the equation if the preload address is not aligned with a cache-line boundary. CWF can offer higher latency for the cache-line load compared to non-CWF read; thus, Intel recommends using the cache-line-aligned addresses for preload.

The preload distance comes out to three cache lines using the memory latency for the PXA27x processor at 200-MHz run mode and 100-MHz SDRAM. This is a rule of thumb for the PXA27x processor for a loop that consumes and produces one cache line.

Correct scheduling of the preload loop can help large memory-based operations such as memory-to-memory copy, page copy, and video operations to load particular image blocks.

5.1.1.2 Preload Loop Limitations

It is not always advantageous to add preloads to a loop. Loop characteristics that limit the value of adding preloads are discussed below.

5.1.1.2.1 Preload Limitations: Throughput bound vs. Latency bound

The worst case is a loop that is bounded by the memory throughput. This does not benefit from preloading because all the system resources to transfer data are quickly allocated and there are no preload instructions that can be executed without impacting (non-preload) memory loads. However, if the application is bounded by the memory latency, preloading can effectively hide the memory latency. Applications requiring large data manipulation, such as graphics applications, video applications etc., can greatly benefit from preloading.

5.1.1.2.2 Preload Limitations: Low Number of Iterations

Loops with a low number of iterations may completely mitigate the advantage of preloading. A loop with a small fixed number of iterations may be faster if the loop is completely unrolled rather than trying to schedule preload instructions.

5.1.1.2.3 Preload Limitations: Bandwidth Consumption

Overuse of preloads can usurp resources and degrade performance because once the bus traffic requests exceed the system resource capacity, the processor stalls. The Intel XScale® Microarchitecture data transfer resources are:

- 4 fill buffers
- 4 pend buffers
- 8 half-cache-line write buffer

SDRAM resources are typically:

- 4 memory banks
- 1 page buffer per bank referencing a 4-KB address range
- 4 transfer request buffers

The following describes how these resources interact. A fill buffer is allocated for each cache read miss. A fill buffer and a pend buffer are allocated for each cache write miss if the memory space is marked as write allocate. A subsequent read to the same cache line does not require a new fill

buffer, but does require a pend buffer and a subsequent write also requires a new pend buffer. A fill buffer is also allocated for each read to a non-cached memory and a write buffer is needed for each memory write to non-cached memory that is non-coalescing. Consequently, an STM instruction listing eight registers and referencing non-cached memory uses eight write buffers assuming they don't coalesce, and two write buffers if they do coalesce. A cache eviction requires a write buffer for each dirty bit set in the cache line. The preload instruction requires a fill buffer for each cache line and 0, 1, or 2 write buffers for an eviction.

When adding preload instructions, ensure that the combination of preload and instruction fetches does not exceed the available system-resource capacity described above or performance is degraded instead of improved. It is important to intersperse preload operations throughout calculations to allow the memory bus traffic to flow freely and to minimize the number of necessary preloads. Section 5.1.1.3 discusses code optimization for preloading.

5.1.1.3 Coding Technique with Preload

Since preloading is a powerful optimization technique, preloading opportunities can be exploited during the code development in the high-level language. The preload instruction can be implemented as a C-callable function and can be used at different places throughout the C-code. Developers can choose to implement two types of routines, one which loads only one cache line or use a function that preloads multiple cache-lines. The data usage (that is linear array striding or 2-D array striding) can influence the choice of the preload scheme. However, only four outstanding preloads are allowed and excessive use must be avoided.

5.1.1.3.1 Coding Technique: Unrolling With Preload

When iterating through a loop, data transfer latency can be hidden by preloading ahead one or more iterations. The solution incurs an unwanted side effect that the final interactions of a loop loads useless data into the cache, polluting the cache, increasing bus traffic and possibly evicting valuable temporal data. This problem can be resolved by preload unrolling. For example:

```
for(i=0; i<NMAX; i++)
{
   prefetch(data[i+2]);
   sum += data[i];
}
```

Iterations i-1 and i preload superfluous data. The problem can be avoid by unrolling the end of the loop.

```
for(i=0; i<NMAX-2; i++)
{
   prefetch(data[i+2]);
   sum += data[i];
}
sum += data[NMAX-2];
sum += data[NMAX-1];
```

Unfortunately, preload loop unrolling does not work on loops with indeterminate iterations.

5.1.1.3.2 Coding Technique: Pointer Preload

Not all looping constructs contain induction variables. However, preloading techniques can still be applied. Refer to this linked list traversal example:

```
while(p) {
   do_something(p->data);
   p = p->next;
}
```

The pointer variable *p* becomes a pseudo-induction variable and the data pointed to by *p->next* can be preloaded to reduce data-transfer latency for the next iteration of the loop. Convert linked lists to arrays as much as possible.

```
while(p) {
   prefetch(p->next);
   do_something(p->data);
   p = p->next;
}
```

Recursive data structure traversal is another construct where pre-loading can be applied. This is similar to linked list traversal. Refer to this preorder traversal of a binary tree:

```
preorder(treeNode *t) {
   if(t) {
      process(t->data);
      preorder(t->left);
      preorder(t->right);
   }
}
```

The pointer variable *t* becomes the pseudo-induction variable in a recursive loop. The data structures pointed to by the values *t->left* and *t->right* can be pre-loaded for the next iteration of the loop.

```
preorder(treeNode *t) {
   if(t) {
      prefetch(t->right);
      prefetch(t->left);
      process(t->data);
      preorder(t->left);
      preorder(t->right);
   }
}
```

The variables are pre-loaded in the opposite order that they are used. If there is a cache conflict and data is evicted from the cache, then only the data from the first pre-load is lost.

5.1.1.3.3 Coding Technique: Preload to Reduce Register Pressure

Preloading can reduce register pressure. When data is needed for an operation, schedule the load far enough in advance to hide the load latency. However, the load ties up the receiving register until the data can be used. For example:

```
   ldr   r2, [r0]
; Process code {not yet cached latency >60 core clocks}
   add   r1, r1, r2
```

In the above case, R2 is unavailable for processing until the add statement. Preloading the data load frees the register for use. The example code becomes:

```
   pld   [r0] ;preload the data keeping r2 available for use
; Process code
   ldr   r2, [r0]
; Process code {ldr result latency is 3 core clocks}
   add   r1, r1, r2
```

With the added preload, register R2 can be used for other operations until just before it is needed.

Apart from code optimization for preload, there are many other techniques (discussed in later chapters) to use while writing C and C++ code.

5.1.2 Array Merging

Stride (the way data structures are walked through) can affect the temporal quality of the data and reduce or increase cache conflicts. The Intel XScale® Microarchitecture data cache has 32 sets of 32 bytes, where each cache line in a set is on a modular 1K-address boundary. It is important to choose data structure sizes and stride requirements that do not overwhelm a given set causing conflicts and increased register pressure. Register pressure can be increased because additional registers are required to track pre-load addresses. This can be achieved by rearranging data structure components to use more parallel access to search and compare elements. Similarly, rearranging data structures so that the sections that are often written fit in the same half cache line[1] can reduce cache eviction write-backs. On a global scale, techniques such as array merging can enhance the spatial locality of the data.

As an example of array merging, refer to this code:

```
int a[NMAX];
int b[NMAX];
int ix;

for (i=0; i<NMAX]; i++)
{
   ix = b[i];
   if (a[i]!= 0)
      ix = a[i];
   do_other calculations;
}
```

In the above code, data is read from both arrays a and b, but a and b are not spatially close. Array merging can place a and b spatially close.

```
struct {
   int a;
   int b;
} c;

int ix;
```

1. A half cache line is 16 bytes for the Intel XScale® Microarchitecture

```
for (i=0; i<NMAX; i++)
{
   ix = c[i].b;
   if (c[i].a != 0)
      ix = c[i].a;
   do_other_calculations;
}
```

As an example of rearranging sections in a structure that are frequently written to, refer to this code sample:

```
struct employee {
   struct employee *prev;
   struct employee *next;
   float Year2DatePay;
   float Year2DateTax;
   int ssno;
   int empid;
   float Year2Date401KDed;
   float Year2DateOtherDed;
};
```

In the data structure shown above, the fields Year2DatePay, Year2DateTax, Year2Date401KDed, and Year2DateOtherDed are likely to change with each pay check. The remaining fields rarely change, however. If the fields are laid out as shown above, assuming that the structure is aligned on a 32-byte boundary, modifications to the Year2Date fields are likely to use two write buffers when the data is written out to memory. However, the number of write buffers that are commonly used can be reduced to 1 by rearranging the fields in the above data structure as shown:

```
struct employee {
   struct employee *prev;
   struct employee *next;
   int ssno;
   int empid;
   float Year2DatePay;
   float Year2DateTax;
   float Year2Date401KDed;
   float Year2DateOtherDed;
};
```

5.1.3 Cache Blocking

Cache blocking techniques, such as strip-mining[1], are used to improve the temporal locality of the data. Given a large data set that can be reused across multiple passes of a loop, data blocking divides the data into smaller chunks that can be loaded into the cache during the first loop and then be available for processing on subsequent loops, thus minimizing cache misses and reducing bus traffic.

As an example of cache blocking refer to this code:

1. Spatially dispersing the data comprising one data set (for example, an array or structure) throughout a memory range instead of keeping the data in contiguous memory locations.

```
for(i=0; i<10000; i++)
   for(j=0; j<10000; j++)
      for(k=0; k<10000; k++)
         C[j][k] += A[i][k] * B[j][i];
```

The variable A[i][k] is completely reused. However, accessing C[j][k] in the j and k loops can displace A[i][j] from the cache. Using cache blocking, the code becomes:

```
for(i=0; i<10000; i++)
   for(j1=0; j<100; j++)
      for(k1=0; k<100; k++)
         for(j2=0; j<100; j++)
            for(k2=0; k<100; k++)
            {
               j = j1 * 100 + j2;
               k = k1 * 100 + k2;
               C[j][k] += A[i][k] * B[j][i];
            }
```

5.1.4 Loop Interchange

As previously mentioned, the sequence in which data is accessed affects cache thrashing. Usually, it is best to access data in a spatially contiguous address range. However, arrays of data may have been laid out such that indexed elements are not physically next to each other. Consider the following C code that places array elements in row major order.

```
for(j=0; j<NMAX; j++)
   for(i=0; i<NMAX; i++)
   {
      prefetch(A[i+1][j]);
      sum += A[i][j];
   }
```

In the above example, A[i][j] and A[i+1][j] are not sequentially next to each other. This situation causes an increase in bus traffic when preloading loop data. In some cases where the loop mathematics are unaffected, the problem can be resolved by induction variable interchange. The above examples becomes:

```
for(i=0; i<NMAX; i++)
   for(j=0; j<NMAX; j++)
   {
      prefetch(A[i][j+1]);
      sum += A[i][j];
   }
```

5.1.5 Loop Fusion

Loop fusion is a process of combining multiple loops, which reuse the same data, into one loop. The advantage of this process is that the reused data is immediately accessible from the data cache. Refer to this example:

```
for(i=0; i<NMAX; i++)
{
   prefetch(A[i+1], c[i+1], c[i+1]);
   A[i] = b[i] + c[i];
}

for(i=0; i<NMAX; i++)
{
   prefetch(D[i+1], c[i+1], A[i+1]);
   D[i] = A[i] + c[i];
}
```

The second loop reuses the data elements A[i] and c[i]. Fusing the loops together produces:

```
for(i=0; i<NMAX; i++)
{
   prefetch(D[i+1], A[i+1], c[i+1], b[i+1]);
   ai = b[i] + c[i];
   A[i] = ai;
   D[i] = ai + c[i];
}
```

In some instances, loop fusion can actually cause performance degradation. In general, loop fusion should only be used when the data operated on in each loop is the same and when all of the contents within the fused loop fit entirely in the instruction cache.

5.1.6 Loop Unrolling

Most compilers unroll fixed length loops when compiled with speed optimizations.

```
for(i=0; i<5; i++){
   f(i);
}
```

is converted to the faster equivalent

```
f(0);
f(1);
f(2);
f(3);
f(4);
```

This optimization eliminates loop overhead.

Additionally, there is also a method of loop unrolling for loops with an unknown amount of iterations at compile time. Therefore, the method is often referred to as *dynamic loop unrolling*. By breaking down an arbitrary size loop into small unrolled blocks, some loop overhead can be avoided.

For example, it is unlikely that a compiler will unroll this code.

```
void f(int nTotalIterations)
{  for(i=0; i<nTotalIterations; i++){
      f(i);}
}
```

Replacing this small loop with the considerably larger code segment below potentially provides a significant performance improvement at the expense of code size.

```
void f(int nTotalIterations)
{
   const int nItersPerBlock = 4;
   int nTotalBlockIters;
   int i;

   // find the largest multiple of nItersPerBlock that is less
   //  than or equal to nTotalIterations
   nTotalBlockIters = (nTotalIterations / nItersPerBlock) *
nItersPerBlock;

   for (i=0; i<nTotalBlockIters; i+=nItersPerBlock)
   {  // unrolling nItersPerBlock times
```

```
        f(i);
        f(i+1);
        f(i+2);
        f(i+3);}

    // any remaining iterations must now be completed
    for (; i<nTotalIterations; ++i)
    {  f(i);}
}
```

Carefully choosing a value for nItersPerBlock based on the task (choosing 8, 16, etc., when large values of nTotalIterations are predicted) increases the benefit of this technique. Again, performance may decline potentially if the instructions within the unrolled block do not fit in the instruction cache. Ensure that all inline functions, inline procedures, and macros used within the block fit within the instruction cache.

5.1.7 Loop Conditionals

Another simple optimization increases the performance of tight loops. When possible, using a decrementing counter that approaches zero can provide a significant performance increase.

For example, here is a typical for() loop.

```
for (i=0; i<1000; ++i)
{  p1();}
```

This code provides the same behavior without as much loop overhead.

```
for (i=1000; i>0; --i)
{  p1(); }
```

For the first loop, most compilers generate an instruction to subtract i from 1000. An instruction is then created to compare the result with 0. In the second loop, a subtraction instruction is not needed, and the 1000 constant does not need to be stored in a register. Freeing that register might save a stack operation for every iteration of the loop, significantly enhancing performance.

5.1.8 If-else versus Switch Statements

Compilers can often generate jump tables for switch statements that can jump to specific code faster than traversing the conditionals of a cascading if-else statement. Some compilers, however, may simply expand the switch statement into a cascading if-else statement. In general, using switch

statements where possible and always placing the most frequently traversed paths higher up in either the switch statement or the cascading if-else code leads to the best optimization by the compiler.

5.1.9 Nested If-Else and Switch Statements

Using nested if-else and switch statements can greatly reduce the number of comparison instructions that are generated by the compiler. For example, consider a switch statement containing 256 case statements. Without knowing if the compiler generates a jump table or a cascading if-else statement, the processor might potentially have to do 256 comparisons only to find that not a single conditional is met.

By breaking the switch into two or more levels, the worst-case lookup is dramatically reduced. Using a switch statement with 16 case statements to jump to 16 other switch statements each with 16 cases reduces the non-existing case lookup to 16 comparisons and the worst-case lookup to 32 comparisons.

5.1.10 Locality in Source Code

On many different levels, code that is cohesive, modular, and decoupled allows compilers to optimize the code to the greatest extent. In C++, these attributes are encouraged by the language. In C, it is very important to keep closely related code and data definitions in the same file as much as possible. The compiler can more efficiently optimize the code this way, and similar data has a higher degree of spatial locality to make better use of the data cache.

5.1.11 Choosing Data Types

Many applications inherently use sub-word data sizes. Packing a set of them into a single word is beneficial for memory accesses and memory bandwidth. Packed data formats can also be processed using Intel® Wireless MMX™ Technology. The Intel XScale® Microarchitecture performs best on word-size data aligned on a 4-byte boundary. Intel® Wireless MMX™ Technology requires data to be aligned on a 8-byte boundary.

5.1.12 Data Alignment For Maximizing Cache Usage

Cache lines begin on 32-byte address boundaries. To maximize cache-line use and minimize cache pollution, align data structures on 32-byte boundaries and sized to a multiple of the cache line sizes. Aligning data structures on cache-address boundaries simplifies later addition of preload instructions to optimize performance.

Not aligning data on cache-line boundaries has the disadvantage of placing the preload address on the corresponding misaligned address. Consider this example:

```
struct {
   long ia;
   long ib;
   long ic;
   long id;
} tdata[IMAX];
```

```
for (i=0; i<IMAX; i++)
{
   PREFETCH(tdata[i+1]);
   tdata[i].ia = tdata[i].ib + tdata[i].ic + tdata[i].id];
   ....
   tdata[i].id = 0;
}
```

In this case, if tdata[] is not aligned to a cache line, then the prefetch using the address of tdata[i+1].ia may not include element id. If the array was aligned on a cache line + 12 bytes, then the prefetch would have to be placed on &tdata[i+1].id.

If the structure is not sized to a multiple of the cache line size, then the preload address must be advanced appropriately and requires extra prefetch instructions. Consider this example:

```
struct {
   long ia;
   long ib;
   long ic;
   long id;
   long ie;
} tdata[IMAX];

ADDRESS predata = &tdata

for (i=0, i<IMAX; i++)
{
   PREFETCH(predata+=16);
   tdata[I].ia =
      tdata[I].ib + tdata[I].ic + tdata[I].id + tdata[I].ie;
   ....
   tdata[I].ie = 0;
}
```

In this case, the preload address was advanced by the size of half a cache line and every other preload instruction is ignored. Further, an additional register is required to track the next preload address.

Generally, not aligning and sizing data adds extra computational overhead.

5.1.13 Placing Literal Pools

The Intel XScale® Microarchitecture does not have a single instruction that can load a literal (a constant or address) to a register; all literals require multiple instructions to load. One technique to load registers with literals in the Intel XScale® Microarchitecture is by loading the literal from a memory location that has been initialized with the constant or address. These blocks of constants are referred to as *literal pools*. These data blocks are located in the text or code address space so that they can be loaded using PC relative addressing. However, references to the literal pool area load the data into the data cache instead of the instruction cache. Therefore, it is possible that the literal may be present in both the data and instruction caches, resulting in wasted space.

For maximum efficiency, the compiler should align all literal pools on cache boundaries and size each pool to a multiple of 32 bytes (the size of a cache line). One additional optimization would be to group frequently used literal pool references into the same cache line. The advantage is that once one of the literals has been loaded, the other seven are available immediately from the data cache.

5.1.14 Global versus Local Variables

The use of variables can impact the efficiency of operation due to additional memory bandwidth requirements. Generally, choosing local variables over global variables provides a greater opportunity to benefit from spatial locality. However, keep local variables to a minimum so that fewer register values must be pushed and subsequently popped from the stack. Also, loops run much more efficiently if all the data operated on can be fit into the registers. Using many variables at once causes memory fetches that decrease performance.

5.1.15 Number of Parameters in Functions

Only the first few (usually the first four) arguments passed into a function are stored in registers. Each additional argument is pushed onto the stack. Keeping the number of parameters in a function equal to or below this amount results in less stack activity over the function call.

5.1.16 Other General Optimizations

- Using the Intel XScale® Microarchitecture mini-data cache to store tables and data that is often used but expensive to calculate enhances performance.
- Limiting the use of unnecessary recursion reduces function-call overhead and memory requirements. Many recursive routines can be converted easily to iterative routines that execute much more quickly using much less memory.
- Avoid library calls and system calls in inner loops.
- For two-dimensional arrays, modifying algorithms to stride in a row-major fashion might offer higher opportunity for preloading and caching opportunities.
- Inlining small functions as much as possible and using macros for commonly executed code sequences reduces the function-call overhead and increases performance.
- Storing data in binary format as opposed to ASCII format reduces the consumption of the limited memory resources found in typical embedded targets.
- Loops often can be terminated without iterating until the conditional is false. Using a break statement to exit the loop as quickly as possible can greatly enhance the loop performance.
- Passing by pointer or reference is highly preferred over passing by value. Only use passing by value when there is a compelling reason to do so. Small data types (4 bytes or less in size) are the exception.

§§

Power Optimization 6

6.1 Introduction

Intel® PXA27x Processor Family (PXA27x processor) includes many features designed for low-power consumption including: five power modes, 1.8 V memory support, and SDRAM auto power down. This chapter describes how to design applications for low-power consumption and the tradeoffs between power consumption and performance.

The major topics covered in this section include considerations for reducing the power consumption of the Intel XScale® core and memory. The power savings and performance tradeoffs vary depending on a user's system configuration.

Refer to the *Intel® PXA270 Processor Electrical, Mechanical, and Thermal Specification* and the *Intel® PXA27x Processor Family Electrical, Mechanical, and Thermal Specification* for detailed information on power consumption.

6.2 Optimizations for Core Power

This section lists optimizations that may help to reduce power consumption consumed primarily by the Intel XScale® core.

6.2.1 Code Optimization for Power Consumption

In most cases, optimizing the operating system (OS) or application for performance also optimizes the code for power consumption. Increasing the caching efficiency and reducing the external memory traffic are the keys to reducing power. To reduce power consumption, implement different assembly level- and high-level language- optimization methods mentioned in previous chapters.

6.2.2 Switching Modes for Saving Power

The PXA27x processor offers multiple power and frequency modes to enable the system to switch between modes, optimizing the system performance and power usage. The operating system needs to develop a power-management strategy based on the system state at different points. In a highly optimized system, the OS can continuously profile the system activities and make the necessary adjustments.

6.2.2.1 Normal Mode

All internal power domains and external power supplies are fully powered and functional. In this mode, all clocks within the processor can be enabled.

6.2.2.2 Idle Mode

This mode is the same as normal mode except the clocks to the CPU are disabled. Recovery is through the assertion of interrupts.

6.2.2.3 Deep Idle Mode

This mode is the same as idle mode except the system bus and all controllers connected to it, operate at 13 MHz. This results in lower power consumption than idle mode. The system bus has very limited bandwidth available in deep-idle mode because it operates at 13 MHz - verify that the system bus bandwidth at 13 MHz is sufficient before implementing deep-idle mode.

Also consider whether peripherals clocked from the core PLL (see "Clocks Manager and Clocks Distribution Block Diagram" in the *Intel® PXA27x Processor Family Developer's Manual*) can properly operate with a 13-MHz clock. For example, the LCD panel might not be able to be refreshed without artifacts using a 13MHz clock.

Deep-idle mode allows CCCR[PPDIS] to remain cleared (CCCR[CPDIS] is always set in deep idle mode). This allows the peripheral units clocked from the peripheral PLL to run at their normal operating frequencies, resulting in these peripherals operating normally. Consider the system bus, however, which is limited to 13-MHz operation. The bandwidth requirements of the entire system (including all peripherals and LCD controller) cannot exceed the bandwidth available on the system bus.

6.2.2.4 Standby Mode

All power domains except VCC_RTC, and VCC_OSC are placed in a low-power mode where state is retained but no activity is allowed, some of the internal power domains (see the *Intel® PXA270 Processor Electrical, Mechanical, and Thermal Specification* and the *Intel® PXA27x Processor Family Electrical, Mechanical, and Thermal Specification*) can be powered off, and both PLLs are disabled; recovery is through external and select internal wake-up events.

6.2.2.5 Sleep Mode

All internal power domains except the VCC_RTC, and VCC_OSC (see the *Intel® PXA270 Processor Electrical, Mechanical, and Thermal Specification* and the *Intel® PXA27x Processor Family Electrical, Mechanical, and Thermal Specification*) can be powered down, all clock sources except to the real time clock and power manager are disabled, and the external-low-voltage-power supplies can be disabled; the remaining power domains are placed in a low-power state where state is retained but no activity is allowed; recovery is through external and select internal wake-up events and recovery requires a system reboot to recover because the program counter is invalid.

6.2.2.6 Deep-Sleep Mode

All internal power domains except the VCC_RTC (real-time controller) and VCC_OSC (oscillator) can be powered down, all clock sources except to the real time clock and power manager are disabled, and the external low-voltage and high-voltage power supplies can be disabled; all power domains are powered directly from the backup battery pin VCC_BATT; the remaining power domains are placed in a low-power state where state is retained but no activity is allowed; recovery is through external and select internal wake-up events and recovery requires a system reboot to recover since the program counter is invalid.

The Intel® PXA27x Processor Family power modes are shown in Table 6-1.

Table 6-1. Power Modes and Typical Power Consumption Summary

Power Mode	Normal	Idle	Deep Idle	Standby	Sleep	Deep Sleep
Usage	Used for normal operation of the core and peripherals.	LCD controller must update the display but there is temporarily no processor computing work.	LCD controller must update the display but there is temporarily no processor computing work. All controllers connected to the system bus use 13MHz clock.	Used for a short-term, low-power "suspend state" when there is no processor computing work and peripherals are stopped.	Typically used for a long-term low-power "off state" when there is no processor computing work and peripherals are stopped.	Typically used for a long-term ultra low-power "off state" when the main battery is low.
					Typically used when device is "switched off" but internal SRAM and external dynamic memory must be preserved.	The RTC and power manager operate from a deeply discharged, unregulated main battery or backup battery
Estimated Power (Measurement Pending)	Refer to the *Intel® PXA27x Processor Family Electrical, Mechanical, and Thermal Specification*					

6.2.3 Wireless Intel Speedstep® Technology Power Manager

Most embedded operating systems do not offer the same power modes that the PXA27x processor has available. It is typical for embedded operating systems to include only two or three power states. The result is that either only a subset of the PXA27x processor power modes can be used in a system or there must be additional software to interact with the OS to map one of the OS power modes to multiple Intel PXA27x processor power modes.

As part of the Wireless Intel Speedstep® Technology Intel has developed a sophisticated software product called the Wireless Intel Speedstep® Technology Power Manager that integrates with the targeted embedded operating systems to allow for all PXA27x processor power modes used by the OS. Through the use of sophisticated system profiling and control software the power manager places the PXA27x processor in the lowest possible power state for the current system state, without any additional OS or application interaction. The effect is to map, through the power-manager software, multiple PXA27x processor low-power modes to single OS power modes.

The power savings realized through the use of Wireless Intel Speedstep® Technology Power Manager can be substantial and are an important part of the Wireless Intel Speedstep® Technology. There are some additional considerations and additions required by applications to take advantage of the power manager, but these additions were minimal.

For details about Wireless Intel Speedstep® Technology Power Manager software, contact an Intel representative.

6.2.4 System Bus Frequency Selection

The system bus runs at a frequency equal to either the run-mode frequency or half the run-mode frequency, depending on whether fast-bus mode is selected. Fast-bus mode is discussed in Section 6.2.4.1, "Fast-Bus Mode".

Selecting the lowest possible run frequency acceptable for applications results in lower core-power consumption.

For example, many combinations of the L, 2N[1] and A bits in the Core Clock Configuration register (CCCR) and B bit in the Clock Configuration register (CLKCFG) result in a core frequency of 208 MHz.

- L=16, N[1]=1.0, A=0, B=0 configures the run frequency as 208 MHz and a turbo frequency of 208 MHz. The system bus runs at 104 MHz and the memory bus runs at 104 MHz.
- L=16, N[1]=1.0, A=0, B=1 configures the run frequency as 208 MHz and a turbo frequency of 208 MHz. The system bus runs at 208 MHz and the memory bus runs at 104 MHz.
- L=16, N[1]=1.0, A=1, B=1 configures the run frequency as 208 MHz and a turbo frequency of 208 MHz. The system bus runs at 208 MHz and the memory bus runs at 208 MHz.
- L=8, N[1]=2.0, A=0, B=0 configures the run frequency as 104 MHz and a turbo frequency of 208 MHz. The system bus runs at 52 MHz and the memory bus runs at 104 MHz.

In all cases, the Intel XScale® core is running at a frequency of 208 MHz. The tradeoffs between the four cases are the speed of the system bus and memory bus versus power consumption. The lower the frequency of either bus, the lower the core-power consumption. However, if an application requires minimal memory latency or high-internal throughput, it might be preferable to run with a higher system-bus frequency or memory bus at the expense of power.

6.2.4.1 Fast-Bus Mode

The system-bus frequency can be doubled through the use of CLKCGF[B] (refer to the "CLKCFG Bit Definitions" table in the *Intel® PXA27x Processor Family Developer's Manual*.) When this bit is set, the system-bus speed is doubled (up to allowable maximum of to 208 MHz) and the system-bus frequency is equal to that of the run-mode frequency. Compared to when fast-bus mode is disabled, this substantially increases the maximum bandwidth of the system bus, but increases the power usage. Refer to the "Fast-Bus Mode" table in the *Intel® PXA27x Processor Family Developer's Manual* for restrictions and detailed information about this mode.

Note: CCCR[L] is limited to 16 in fast-bus mode.

6.2.4.2 Half-Turbo Mode

Systems that have high bus-usage requirements but low Intel XScale® core frequency requirements can benefit from the use of half-turbo mode. This mode causes the Intel XScale® core to be clocked at half the selected run-mode frequency. Only the core frequency is affected by this mode. The bus speed configurations are unaffected in this mode. By using this mode to select a lower core frequency, the power used by the core is minimized. Ensure that the Intel XScale® core frequency is sufficient for the loading on the core; otherwise, system performance is degraded.

Refer to the "CLKCFG Bit Definitions" table in the *Intel® PXA27x Processor Family Developer's Manual* for information on the bit settings. Refer to the "Half Turbo Mode" table in the *Intel® PXA27x Processor Family Developer's Manual* for restrictions and detailed information about this mode.

1. The value of N used to determine the Turbo Mode frequency is one half the value written to CCCR[2N]

6.3 Optimizations for Memory and Peripheral Power

This section lists optimizations that may help reduce power consumption primarily by the memory and peripheral subsystems. Using hardware that supports lower voltages such as 1.8 V SDRAM flash is a key factor.

6.3.1 Improved Caching and Internal Memory Usage

Transactions to the external memory are the largest contributors to the system power consumption. Reducing external memory transactions—by improving the cache efficiency and internal-SRAM efficiency—reduces the power consumption.

6.3.2 SDRAM Auto Power Down (APD)

The APD bit is a setting in the memory controller MDREFR register that allows the clock-enable and clocks to SDRAM or synchronous static memory to be disabled automatically when not used. When the memory does need to be accessed, an additional latency of one clock occurs to re-enable the clock signals.

Enabling APD is useful during periods of system inactivity, such as the idle loop. During periods of system activity, weigh the additional power savings with APD against the additional latency incurred.

6.3.3 External Memory Bus Buffer Strength Registers

The output drivers for the PXA27x processor external-memory bus have programmable-strength settings. This feature allows for simple, software-based control of the output-driver impedance for the external-memory bus. Use these registers to match the driver strength of the PXA27x processor to external-memory bus. Set the buffer strength to the lowest possible setting (minimum drive strength) that still allows for reliable memory-system performance. This minimizes the power usage of the external memory bus, which is a major component of total system power. Refer to the Programmable Output Buffer Strength registers in the *Intel® PXA27x Processor Family Developer's Manual* for more information.

6.3.4 Peripheral Clock Gating

If peripherals are not used, their clocks can be disabled to save power. The CKEN register in the clocks manager contain configuration bits that control the clocks to each peripheral. To save power, disable unnecessary peripherals.

6.3.5 LCD Subsystem

The LCD subsystem, which consists of the LCD panel, power supplies, and backlight, can be optimized to reduce system power. Lowering the refresh rate of the LCD panel reduces power used by the memory bus and can be done dynamically by monitoring the system inactivity and lowering the refresh rate accordingly. If the LCD has a backlight, disabling it after a short time further reduces power. Software can disable the LCD during long periods of inactivity and enter sleep mode.

Another way to save power is to reduce memory traffic. Instead of using a color depth of 16 bits per pixel, systems could use 8 bits per pixel to reduce the memory bandwidth for LCD refresh.

The output drivers for the PXA27x processor LCD data pins have programmable strength settings. This feature allows for simple, software-based control of the output-driver impedance for the data pins on the LCD bus. Use these registers to match the driver strength of the PXA27x processor to LCD bus. Set the buffer strength to the lowest possible setting (minimum drive strength) that still allows for reliable bus performance. This minimizes the power usage of the LCD data pins, which can be a major component of total system power, since in a typical system, these pins are always driven while the screen is on. Refer to the "LCD Buffer Strength Control Register (LCDBSCNTR)" table in the *Intel® PXA27x Processor Family Developer's Manual* for more information.

6.3.6 Voltage and Regulators

The PXA27x processor supports 1.8 V memory. Depending on the memory usage of applications, there may be significant power savings using 1.8 V SDRAM compared to 2.5 V or 3.3 V SDRAM. If no other devices in the system use 1.8 V, then consider the power savings compared to the extra components and board real-estate to support 1.8 V.

6.3.7 Operating Mode Recommendations for Power Savings

6.3.7.1 Normal Mode

It may require less power to run at a higher Intel XScale® core frequency/voltage to complete a task and then drop the operating frequency and Intel XScale® core voltage than to run at a constant frequency and voltage. Profile the OS and applications to determine the optimum Intel XScale® core operating frequency. Use the lowest possible Intel XScale® core voltage to run at the required frequency.

For lowest power consumption in normal mode:

- Use Intel® Integrated Performance Primitives
- Use Wireless Intel Speedstep® Technology
- Use Intel® Power Manager
- Use low-power modes when possible
- Enable both caches
- Use a caching policy of read allocate/write back when possible
- Set the APD bit (see Section 6.3.2)
- Configure the System Memory Buffer Strength Control registers for the minimal setting for reliable memory bus performance.
- Configure the LCD Buffer Strength Control register for the minimal setting for reliable LCD bus performance.

6.3.7.2 Idle Mode

Use idle mode only for brief periods of time when the Intel XScale® core is soon required to perform calculations.

6.3.7.3 Deep-Idle Mode

Use deep-idle mode instead of idle mode whenever the time required without Intel XScale® core operation is long enough to accomplish the necessary voltage and frequency changes.

6.3.7.4 Standby Mode

For lowest power consumption in standby mode:

- Ensure VCC_Core and VCC_SRAM are 1.1V.
- Configure all possible I/O pins as outputs and drive them to a logic low.
- Drive TDI and TMS to a logic high.
- Drive the USB client differential inputs (USBCP and USBCN) to a logic high.

6.3.7.5 Sleep Mode

For lowest power consumption in sleep mode:

- Disable and ground VCC_SRAM, VCC_Core and VCC_PLL.
- Configure all possible IO pins as outputs and drive them to a logic low.
- Drive TDI and TMS to a logic high.
- Drive the USB client differential inputs (USBCP and USBCN) to a logic high.
- Enable the DC-DC converter, if possible.

6.3.7.6 Deep-Sleep Mode

For lowest power consumption in deep-sleep mode:

- Disable and ground all non-battery voltage supplies.
- Configure all possible I/O pins as outputs and drive them to a logic low.
- Drive TDI and TMS to a logic high.
- Drive the USB client differential inputs (USBCP and USBCN) to a logic high.
- Enable the DC-DC converter, if possible.

§§

Performance Checklist A

A.1 Performance Optimization Tips

- Use fast 100-MHz SDRAM with a CAS latency of 2.
- Use flash that supports read bursts of four or eight.
- Optimize flash and SDRAM timings.
- Don't set the LCD refresh higher than required.
- When possible, use DMA in device drivers instead of programmed I/O to load/unload FIFOs.
- Use interrupts, avoid polling, and frequent interrupts.
- Lock cache lines and TLB entries for frequently used operations.
- Make sure the instruction and data caches are enabled.
- Use cache policies to optimize throughput and performance.
- Use the internal SRAM.
- Park the system bus arbiter on the Intel XScale® core unless performing task which heavily uses a different system bus client.
- Make the LCD frame buffer non-cached but bufferable.
- Use write-back caches if possible.
- Optimize assembly code based on the suggestions presented in this guide.
- Enable the branch target buffer.
- Configure non cacheable memory as bufferable whenever possible.
- Choose a fast run mode speed for minimal memory latency and a fast turbo mode for processing speed.
- Optimize data loads with the PLD instruction and data stores with code that encourages write coalescing.
- Choose a V5TE compiler optimized for Intel XScale® Microarchitecture.
- Use the optimized Intel® Integrated Performance Primitives.
- Enable the Intel® Wireless MMX™ Technology media co-processor.
- Include all necessary software hooks for Intel VTune™ environment.
- Use LCD color conversion hardware for video applications.
- Use fast-bus mode for high system-bus bandwidth.
- Use alternate memory clock setting for high memory bus throughput.
- Ensure that the latest board support package is being used.
- Use Intel® Quick Capture Interface for image data.

A.2 Power Optimization Guidelines

- Use 1.8 V SDRAM, if possible.
- Implement multiple power modes so that the system can switch to the lowest possible power mode required to meet the processing requirements at that time.
- Switch to run mode instead of turbo mode during times when less processing power is needed at the lowest possible run and turbo mode frequencies.
- Higher run and turbo-mode frequencies consume more power. Optimize system for desired power and performance.
- Consider performing a frequency change sequence to a lower run frequency if the 500-µS delay is acceptable for your application.
- Use the SDRAM APD bit.
- Use half-turbo mode to reduce the core clock frequency without impacting bus operations.
- Use minimum Memory Buffer register strength settings possible.
- Use minimum LCD Buffer Strength register settings possible.
- Use Intel® Quick Capture Interface to bring image data in YCbCr mode, when possible.


ARM
POWERED

Glossary

3G An industry term used to describe the next, still-to-come generation of wireless applications. It represents a move from circuit-switched communications (where a device user has to dial in to a network) to broadband, high-speed, packet-based wireless networks (which are always on). The first generation of wireless communications relied on analog technology, followed by digital wireless communications. The third generation expands the digital capabilities by including high-speed connections and increased reliability.

802.11 Wireless specifications developed by the IEEE, outlining the means to manage packet traffic over a network and ensure that packets do not collide, which could result in the loss of data, when travelling from device to device.

8PSK 8 phase shift key modulation scheme. Used in the EDGE standard.

AC ’97 AC-link standard serial interface for modem and audio

ACK Handshake packet indicating a positive acknowledgment.

Active device A device that is powered and is not in the suspended state.

Air interface the RF interface between a mobile cellular handset and the base station

AMPS Advanced Mobile Phone Service. A term used for analog technologies, the first generation of wireless technologies.

Analog Radio signals that are converted into a format that allows them to carry data. Cellular phones and other wireless devices use analog in geographic areas with insufficient digital networks.

ARM* V5te An ARM* architecture designation indicating the processor is conforms to ARM* architecture version 5, including “Thumb” mode and the “El Segundo” DSP extensions.

Asynchronous Data Data transferred at irregular intervals with relaxed latency requirements.

Asynchronous RA The incoming data rate, Fs i, and the outgoing data rate, Fs o, of the RA process are independent (i.e., there is no shared master clock). See also rate adaptation.

Asynchronous SRC The incoming sample rate, Fsi, and outgoing sample rate, Fso, of the SRC process are independent (i.e., there is no shared master clock). See also sample rate conversion.

Audio device A device that sources or sinks sampled analog data.

AWG# The measurement of a wire’s cross-section, as defined by the American Wire Gauge standard.

Babble Unexpected bus activity that persists beyond a specified point in a (micro)frame.

Backlight Inverter A device to drive cold cathode fluorescent lamps used to illuminate LCD panels.

Bandwidth The amount of data transmitted per unit of time, typically bits per second (b/s) or bytes per second (B/s). The size of a network “pipe” or channel for communications in wired networks. In wireless, it refers to the range of available frequencies that carry a signal.

Base Station The telephone company’s interface to the Mobile Station

BGA Ball Grid Array

BFSK binary frequency shift keying. A coding scheme for digital data.

Bit A unit of information used by digital computers. Represents the smallest piece of addressable memory within a computer. A bit expresses the choice between two possibilities and is typically represented by a logical one (1) or zero (0).

Bit Stuffing Insertion of a “0” bit into a data stream to cause an electrical transition on the data wires, allowing a PLL to remain locked.

Blackberry A two-way wireless device (pager) made by Research In Motion (RIM) that allows users to check e-mail and voice mail translated into text, as well as page other users of a wireless network service. It has a miniature “qwerty” keyboard that can be used by your thumbs, and uses SMS protocol. A Blackberry user must subscribe to the proprietary wireless service that allows for data transmission.

Bluetooth A short-range wireless specification that allows for radio connections between devices within a 30-foot range of each other. The name comes from 10th-century Danish King Harald Blatand (Bluetooth), who unified Denmark and Norway.

BPSK binary phase shift keying. A means of encoding digital data into a signal using phase-modulated communications.

b/s Transmission rate expressed in bits per second.

B/s Transmission rate expressed in bytes per second.

BTB Branch Target Buffer

BTS Base Transmitter Station

Buffer Storage used to compensate for a difference in data rates or time of occurrence of events, when transmitting data from one device to another.

Bulk Transfer One of the four USB transfer types. Bulk transfers are non-periodic, large bursty communication typically used for a transfer that can use any available bandwidth and can also be delayed until bandwidth is available. See also transfer type.

Bus Enumeration Detecting and identifying USB devices.

Byte A data element that is eight bits in size.

Capabilities Those attributes of a USB device that are administrated by the host.

CAS Cycle Accurate Simulator

CAS-B4-RAS See CBR.

CBR CAS Before RAS. Column Address Strobe Before Row Address Strobe. A fast refresh technique in which the DRAM keeps track of the next row it needs to refresh, thus simplifying what a system would have to do to refresh the part.

CDMA Code Division Multiple Access U.S. wireless carriers Sprint PCD and Verizon use CDMA to allocate bandwidth for users of digital wireless devices. CDMA distinguishes between multiple transmissions carried simultaneously on a single wireless signal. It carries the transmissions on that signal, freeing network room for the wireless carrier and providing interference-free calls for the user. Several versions of the standard are still under development. CDMA should increase network capacity for wireless carriers and improve the quality of wireless messaging. CDMA is an alternative to GSM.

CDPD Cellular Digital Packet Data Telecommunications companies can use DCPD to transfer data on unused cellular networks to other users. IF one section, or “cell” of the network is overtaxed, DCPD automatically allows for the reallocation of services.

Cellular Technology that senses analog or digital transmissions from transmitters that have areas of coverage called cells. As a user of a cellular phone moves between transmitters from one cell to another, the users’ call travels from transmitter to transmitter uninterrupted.

Circuit Switched Used by wireless carriers, this method lets a user connect to a network or the Internet by dialing in, such as with a traditional phone line. Circuit switched connections are typically slower and less reliable than packet-switched networks, but are currently the primary method of network access for wireless users in the U.S.

CF Compact Flash memory and I/O card interface

Characteristics Those qualities of a USB device that are unchangeable; for example, the device class is a device characteristic.

Client Software resident on the host that interacts with the USB System Software to arrange data transfer between a function and the host. The client is often the data provider and consumer for transferred data.

CML Current mode logic

Configuring Software Software resident on the host software that is responsible for configuring a USB device. This may be a system configuration or software specific to the device.

Control Endpoint A pair of device endpoints with the same endpoint number that are used by a control pipe. Control endpoints transfer data in both directions and, therefore, use both endpoint directions of a device address and endpoint number combination. Thus, each control endpoint consumes two endpoint addresses.

Control Pipe Same as a message pipe.

Control Transfer One of the four USB transfer types. Control transfers support configuration/command/status type communications between client and function. See also transfer type.

CRC See Cyclic Redundancy Check.

CSP Chip Scale Package.

CTE Coefficient of thermal expansion

CTI Computer Telephony Integration.

Cyclic Redundancy Check (CRC) A check performed on data to see if an error has occurred in transmitting, reading, or writing the data. The result of a CRC is typically stored or transmitted with the checked data. The stored or transmitted result is compared to a CRC calculated for the data to determine if an error has occurred.

D-cache Data cache

DECT the Digital European Cordless Telecommunications standard

Default Address An address defined by the USB Specification and used by a USB device when it is first powered or reset. The default address is 00H.

Default Pipe The message pipe created by the USB System Software to pass control and status information between the host and a USB device’s endpoint zero.

Device A logical or physical entity that performs a function. The actual entity described depends on the context of the reference. At the lowest level, “device” may refer to a single hardware component, as in a memory device. At a higher level, it may refer to a collection of hardware components that perform a particular function, such as a USB interface device. At an even higher level, device may refer to the function performed by an entity attached to the USB; for example, a data/FAX modem device. Devices may be physical, electrical, addressable, and logical. When used as a non-specific reference, a USB device is either a hub or a function.

Device Address A seven-bit value representing the address of a device on the USB. The device address is the default address (00H) when the USB device is first powered or the device is reset. Devices are assigned a unique device address by the USB System Software.

Device Endpoint A uniquely addressable portion of a USB device that is the source or sink of information in a communication flow between the host and device. See also endpoint address.

Device Resources Resources provided by USB devices, such as buffer space and endpoints. See also Host Resources and Universal Serial Bus Resources.

Device Software Software that is responsible for using a USB device. This software may or may not also be responsible for configuring the device for use.

DMA Direct Memory Access

Downstream The direction of data flow from the host or away from the host. A downstream port is the port on a hub electrically farthest from the host that generates downstream data traffic from the hub. Downstream ports receive upstream data traffic.

DQPSK Differential Quadrature Phase Shift Keying a modulation technique used in TDMA.

Driver When referring to hardware, an I/O pad that drives an external load. When referring to software, a program responsible for interfacing to a hardware device, that is, a device driver.

DSP Digital Signal Processing

DSTN Passive LCD Panel.

Dual band mobile phone A phone that supports both analog and digital technologies by picking up analog signals when digital signals fade. Most mobile phones are not dual-band.

DWORD Double word. A data element that is two words (i.e., four bytes or 32 bits) in size.

Dynamic Insertion and Removal The ability to attach and remove devices while the host is in operation.

E2PROM See Electrically Erasable Programmable Read Only Memory.

EAV End of active video

EDGE Enhanced Data GSM Environment. A faster version of the GSM standard. It is faster because it can carry messages using broadband networks that employ more bandwidth than standard GSM networks.

EEPROM See Electrically Erasable Programmable Read Only Memory.

Electrically Erasable Programmable Read Only Memory (EEPROM) Non-volatile re-writable memory storage technology.

End User The user of a host.

Endpoint See device endpoint.

Endpoint Address The combination of an endpoint number and an endpoint direction on a USB device. Each endpoint address supports data transfer in one direction.

Endpoint Direction The direction of data transfer on the USB. The direction can be either IN or OUT. IN refers to transfers to the host; OUT refers to transfers from the host.

Endpoint Number A four-bit value between 0H and FH, inclusive, associated with an endpoint on a USB device.

Envelope detector An electronic circuit inside a USB device that monitors the USB data lines and detects certain voltage related signal characteristics.

EOF End-of-(micro)Frame.

EOP End-of-Packet.

EOTD Enhanced Observed Time Difference

ETM Embedded Trace Macrocell, the ARM* real-time trace capability

External Port See port.

Eye pattern A representation of USB signaling that provides minimum and maximum voltage levels as well as signal jitter.

FAR Fault Address Register, part of the ARM* architecture.

False EOP A spurious, usually noise-induced event that is interpreted by a packet receiver as an EOP.

FDD The Mobile Station transmits on one frequency; the Base Station transmits on another frequency

FDM Frequency Division Multiplexing. Each Mobile station transmits on a different frequency (within a cell).

FDMA Frequency Division Multiple Access. An analog standard that lets multiple users access a group of radio frequency bands and eliminates interference of message traffic.

FHSS See Frequency Hopping Spread Spectrum.

FIQ Fast Interrupt Request. See Interrupt Request.

Frame A 1 millisecond time base established on full-/low-speed buses.

Frame Pattern A sequence of frames that exhibit a repeating pattern in the number of samples transmitted per frame. For a 44.1 kHz audio transfer, the frame pattern could be nine frames containing 44 samples followed by one frame containing 45 samples.

Frequency Hopping Spread Spectrum A method by which a carrier spreads out packets of information (voice or data) over different frequencies. For example, a phone call is carried on several different frequencies so that when one frequency is lost another picks up the call without breaking the connection.

Fs See sample rate.

FSR Fault Status Register, part of the ARM* architecture.

Full-duplex Computer data transmission occurring in both directions simultaneously.

Full-speed USB operation at 12 Mb/s. See also low-speed and high-speed.

Function A USB device that provides a capability to the host, such as an ISDN connection, a digital microphone, or speakers.

GMSK Gaussian Minimum Shift Keying. A modulation scheme used in GSM.

GPRS General Packet Radio Service A technology that sends packets of data across a wireless network at speeds up to 114 Kbps. Unlike circuit-switched networks, wireless users do not have to dial in to networks to download information; GPRS wireless devices are “always on” in that they can send and receive data without dial-ins. GPRS works with GSM.

GPS Global Positioning Systems

GSM Global System for Mobile Communications. A standard for how data is coded and transferred through the wireless spectrum. The European wireless standard, also used in parts of Asia, GSM is an alternative to CDMA. GSM digitizes and compresses data and sends it across a channel with two other streams of user data. GSM is based on TDMA technology.

Hamming Distance The distance (number of bits) between encoded values that can change without causing a decode into the wrong value.

Handshake Packet A packet that acknowledges or rejects a specific condition. For examples, see ACK and NAK.

HDML Handheld Device Markup Language. HDML uses hypertext transfer protocol (HTTP) to display text versions of web pages on wireless devices. Unlike WML, HDML is not based on XML. HDML does not allow scripts, while WML uses a variant of JavaScript. Web site developers using HDML must re-code their web pages in HDML to be viewed on the smaller screen sizes of handheld devices.

HARP Windows CE standard development platform spec (Hardware Adaptation Reference Platform)

High-bandwidth endpoint A high-speed device endpoint that transfers more than 1024 bytes and less than 3073 bytes per microframe.

High-speed USB operation at 480 Mb/s. See also low-speed and full-speed.

Host The host computer system where the USB Host controller is installed. This includes the host hardware platform (CPU, bus, and so forth.) and the operating system in use.

Host Controller The host’s USB interface.

Host Controller Driver (HCD) The USB software layer that abstracts the Host controller hardware. The Host controller driver provides an SPI for interaction with a Host controller. The Host controller driver hides the specifics of the Host controller hardware implementation.

Host Resources Resources provided by the host, such as buffer space and interrupts. See also **Device Resources** and **Universal Serial Bus Resources**.

HSTL High-speed transceiver logic

Hub A USB device that provides additional connections to the USB.

Hub Tier One plus the number of USB links in a communication path between the host and a function.

IMMU Instruction Memory Management Unit, part of the Intel XScale® core.

I-Mode A Japanese wireless service for transferring packet-based data to handheld devices created by NTT DoCoMo. I-Mode is based on a compact version of HTML and does not currently use WAP.

I-cache Instruction cache

IBIS I/O Buffer Information Specification is a behavioral description of the I/O buffers and package characteristics of a semiconductor device. IBIS models use a standard format to make it easier to import data into circuit simulation software packages.

iDEN Integrated Digital Enhanced Network. A technology that allows users to access phone calls, two-way radio transmissions, paging and data transmissions from one wireless device. iDEN was developed by Motorola and based on TDMA.

Interrupt Request (IRQ) A hardware signal that allows a device to request attention from a host. The host typically invokes an interrupt service routine to handle the condition that caused the request.

Interrupt Transfer One of the four USB transfer types. Interrupt transfer characteristics are small data, non-periodic, low-frequency, and bounded-latency. Interrupt transfers are typically used to handle service needs. See also transfer type.

I/O Request Packet An identifiable request by a software client to move data between itself (on the host) and an endpoint of a device in an appropriate direction.

IrDA Infrared Development Association

IRP See I/O Request Packet.

IRQ See Interrupt Request.

ISI Inter-signal interference. Data ghosting caused when multi-path delay causes previous symbols to interfere with the one currently being processed.

ISM Industrial, Scientific, and Medical band. Part of the wireless spectrum that is less regulated, such as 802.11.

Isochronous Data A stream of data whose timing is implied by its delivery rate.

Isochronous Device An entity with isochronous endpoints, as defined in the USB Specification, that sources or sinks sampled analog streams or synchronous data streams.

Isochronous Sink Endpoint An endpoint that is capable of consuming an isochronous data stream that is sent by the host.

Isochronous Source Endpoint An endpoint that is capable of producing an isochronous data stream and sending it to the host.

Isochronous Transfer One of the four USB transfer types. Isochronous transfers are used when working with isochronous data. Isochronous transfers provide periodic, continuous communication between host and device. See also transfer type.

Jitter A tendency toward lack of synchronization caused by mechanical or electrical changes. More specifically, the phase shift of digital pulses over a transmission medium.

kb/s Transmission rate expressed in kilobits per second. A measurement of bandwidth in the U.S.

kB/s Transmission rate expressed in kilobytes per second.

Little endian Method of storing data that places the least significant byte of multiple-byte values at lower storage addresses. For example, a 16-bit integer stored in little endian format places the least significant byte at the lower address and the most significant byte at the next address.

LOA Loss of bus activity characterized by an SOP without a corresponding EOP.

Low-speed USB operation at 1.5 Mb/s. See also full-speed and high-speed.

LSb Least significant bit.

LSB Least significant byte.

LVDS Low-voltage differential signal

MAC Multiply Accumulate unit

Mb/s Transmission rate expressed in megabits per second.

MB/s Transmission rate expressed in megabytes per second.

MC Media Center. A combination digital set-top box, video and music jukebox, personal video recorder and an Internet gateway and firewall that hooks up to a broadband connection.

Message Pipe A bidirectional pipe that transfers data using a request/data/status paradigm. The data has an imposed structure that allows requests to be reliably identified and communicated.

Microframe A 125 microsecond time base established on high-speed buses.

MMC Multimedia Card - small form factor memory and I/O card

MMX Technology The Intel® MMX™ technology comprises a set of instructions that are designed to greatly enhance the performance of advanced media and communications applications. See chapter 10 of the *Intel Architecture Software Developers Manual, Volume 3: System Programming Guid*e, Order #245472.

Mobile Station Cellular Telephone handset

M-PSK multilevel phase shift keying. A convention for encoding digital data in which there are multiple states.

MMU Memory Management Unit, part of the Intel XScale®core.

MSb Most significant bit.

MSB Most significant byte.

MSL Mobile Scalable Link.

NAK Handshake packet indicating a negative acknowledgment.

Non Return to Zero Invert (NRZI) A method of encoding serial data in which ones and zeroes are represented by opposite and alternating high and low voltages where there is no return to zero (reference) voltage between encoded bits. Eliminates the need for clock pulses.

NRZI See Non Return to Zero Invert.

Object Host software or data structure representing a USB entity.

OFDM See Orthogonal Frequency Division Multiplexing.

Orthogonal Frequency Division Multiplexing A special form of multi-carrier modulation. In a multi-path channel, most conventional modulation techniques are sensitive to inter-symbol interference unless the channel symbol rate is small compared to the delay spread of the channel. OFDM is significantly less sensitive to inter-symbol interference, because a special set of signals is used to build the composite transmitted signal. The basic idea is that each bit occupies a frequency-time window that ensures little or no distortion of the waveform. In practice, it means that bits are transmitted in parallel over a number of frequency-nonselective channels.

Packet A bundle of data organized in a group for transmission. Packets typically contain three elements: control information (for example, source, destination, and length), the data to be transferred, and error detection and correction bits. Packet data is the basis for packet-switched networks, which eliminate the need to dial-in to send or receive information, because they are "always on."

Packet Buffer The logical buffer used by a USB device for sending or receiving a single packet. This determines the maximum packet size the device can send or receive.

Packet ID (PID) A field in a USB packet that indicates the type of packet, and by inference, the format of the packet and the type of error detection applied to the packet.

Packet Switched Network Networks that transfer packets of data.

PCMCIA Personal Computer Memory Card Interface Association (PC Card)

PCS Personal Communications services. An alternative to cellular, PCD works like cellular technology because it sends calls from transmitter to transmitter as a caller moves. But PCS uses its own network, not a cellular network, and offers fewer "blind spots" than cellular, where calls are not available. PCS transmitters are generally closer together than their cellular counterparts.

PDA Personal Digital Assistant. A mobile handheld device that gives users access to text-based information. Users can synchronize their PDAs with a PC or network; some models support wireless communication to retrieve and send e-mail and get information from the Internet.

Phase A token, data, or handshake packet. A transaction has three phases.

Phase Locked Loop (PLL) A circuit that acts as a phase detector to keep an oscillator in phase with an incoming frequency.

Physical Device A device that has a physical implementation; for example, speakers, microphones, and CD players.

PID See Packet ID or Process ID.

PIO Programmed input/output

Pipe A logical abstraction representing the association between an endpoint on a device and software on the host. A pipe has several attributes; for example, a pipe may transfer data as streams (stream pipe) or messages (message pipe). See also stream pipe and message pipe.

PLL See Phase Locked Loop.

PM Phase Modulation.

Polling Asking multiple devices, one at a time, if they have any data to transmit.

POR See Power On Reset.

Port Point of access to or from a system or circuit. For the USB, the point where a USB device is attached.

Power On Reset (POR) Restoring a storage device, register, or memory to a predetermined state when power is applied.

Process ID Process identifier

Programmable Data Rate Either a fixed data rate (single-frequency endpoints), a limited number of data rates (32 kHz, 44.1 kHz, 48 kHz, …), or a continuously programmable data rate. The exact programming capabilities of an endpoint must be reported in the appropriate class-specific endpoint descriptors.

Protocol A specific set of rules, procedures, or conventions relating to format and timing of data transmission between two devices.

PSP Programmable Serial Protocol

PWM Pulse Width Modulator

QBS Qualification By Similarity. A technique allowed by JEDEC for part qualification when target parameters are fully understood and data exist to warrant omitting a specific test.

QAM quadrature amplitude modulation. A coding scheme for digital data.

QPSK quadrature phase shift keying. A convention for encoding digital data into a signal using phase-modulated communications.

RA See rate adaptation.

Radio Frequency Device These devices use radio frequencies to transmit data. One typical use is for bar code scanning of products in a warehouse or distribution center, and sending that information to an ERP database.

Rate Adaptation The process by which an incoming data stream, sampled at Fs i, is converted to an outgoing data stream, sampled at Fs o, with a certain loss of quality, determined by the rate adaptation algorithm. Error control mechanisms are required for the process. Fs i and Fs o can be different and asynchronous. Fs i is the input data rate of the RA; Fs o is the output data rate of the RA.

Request A request made to a USB device contained within the data portion of a SETUP packet.

Retire The action of completing service for a transfer and notifying the appropriate software client of the completion.

RGBT Red, Green, Blue, Transparency

ROM Read Only Memory.

Root Hub A USB hub directly attached to the Host controller. This hub (tier 1) is attached to the host.

Root Port The downstream port on a Root Hub.

RTC Real-Time Clock

SA-1110 StrongARM* based applications processor for handheld products

Intel® StrongARM* SA-1111 Companion chip for the Intel® SA-1110 processor

SAD Sum of absolute differences

Sample The smallest unit of data on which an endpoint operates; a property of an endpoint.

Sample Rate (Fs) The number of samples per second, expressed in Hertz (Hz).

Sample Rate Conversion (SRC) A dedicated implementation of the RA process for use on sampled analog data streams. The error control mechanism is replaced by interpolating techniques. Service A procedure provided by a System Programming Interface (SPI).

Satellite Phone Phones that connect callers by satellite. Users have a world-wide alternative to terrestrial connections. Typical use is for isolated users, such as crews of deep-see oil rigs with phones configured to connect to a satellite service.

SAV Start of active video

SAW Surface Acoustic Wave filter

SDRAM Synchronous Dynamic Random Access Memory.

Service Interval The period between consecutive requests to a USB endpoint to send or receive data.

Service Jitter The deviation of service delivery from its scheduled delivery time.

Service Rate The number of services to a given endpoint per unit time.

SIMD Single Instruction Multiple Data (a parallel processing architecture).

Smart Phone A combination of a mobile phone and a PDA, which allow users to communicate as well as perform tasks; such as, accessing the Internet and storing contacts in a database. Smart phones have a PDA-like screen.

SMROM Synchronous Mask ROM

SMS Short Messaging Service. A service through which users can send text-based messages from one device to another. The message can be up to 160 characters and appears on the screen of the receiving device. SMS works with GSM networks.

SOC System On Chip

SOF See Start-of-Frame.

SOP Start-of-Packet.

SPI See System Programming Interface. Also, “Serial Peripheral Interface protocol.

SPI Serial Peripheral Interface

Split transaction A transaction type supported by host controllers and hubs. This transaction type allows full- and low-speed devices to be attached to hubs operating at high-speed.

Spread Spectrum An encoding technique patented by actress Hedy Lamarr and composer George Antheil, which broadcasts a signal over a range of frequencies.

SRAM Static Random Access Memory.

SRC See Sample Rate Conversion.

SSE Streaming SIMD Extensions

SSE2 Streaming SIMD Extensions 2: for Intel Architecture machines, 144 new instructions, a 128-bit SIMD integer arithmetic and 128-bit SIMD double precision floating point instructions, enabling enhanced multimedia experiences.

SSP Synchronous Serial Port

SSTL Stub series terminated logic

Stage One part of the sequence composing a control transfer; stages include the Setup stage, the Data stage, and the Status stage.

Start-of-Frame (SOF) The first transaction in each (micro)frame. An SOF allows endpoints to identify the start of the (micro)frame and synchronize internal endpoint clocks to the host.

Stream Pipe A pipe that transfers data as a stream of samples with no defined USB structure

SWI Software interrupt.

Synchronization Type A classification that characterizes an isochronous endpoint's capability to connect to other isochronous endpoints.

Synchronous RA The incoming data rate, Fsi, and the outgoing data rate, Fso, of the RA process are derived from the same master clock. There is a fixed relation between Fsi and Fso.

Synchronous SRC The incoming sample rate, Fsi, and outgoing sample rate, Fso, of the SRC process are derived from the same master clock. There is a fixed relation between Fsi and Fso.

System Programming Interface (SPI) A defined interface to services provided by system software.

TC Temperature Cycling

TDD Time Division Duplexing The Mobile Station and the Base Station transmit on same frequency at different times.

TDM See Time Division Multiplexing.

TDMA Time Division Multiple Access. TDMA protocol allows multiple users to access a single radio frequency by allocating time slots for use to multiple voice or data calls. TDMA breaks down data transmissions, such as a phone conversation, into fragments and transmits each fragment in a short burst, assigning each fragment a time slot. With a cell phone, the caller would not detect this fragmentation. TDMA works with GSM and digital cellular services.

TDR See Time Domain Reflectometer.

Termination Passive components attached at the end of cables to prevent signals from being reflected or echoed.

TFT Thin Film Twist, a type of active LCD panel.

Three-state a high-impedance state in which the output is floating and is electrically isolated from the buffer's circuitry.

Time Division Multiplexing (TDM) A method of transmitting multiple signals (data, voice, and/or video) simultaneously over one communications medium by interleaving a piece of each signal one after another.

Time Domain Reflectometer (TDR) An instrument capable of measuring impedance characteristics of the USB signal lines.

Time-out The detection of a lack of bus activity for some predetermined interval.

Token Packet A type of packet that identifies what transaction is to be performed on the bus.

TPV Third Party Vendor

Transaction The delivery of service to an endpoint; consists of a token packet, optional data packet, and optional handshake packet. Specific packets are allowed/required based on the transaction type.

Transaction translator A functional component of a USB hub. The Transaction Translator responds to special high-speed transactions and translates them to full/low-speed transactions with full/low-speed devices attached on downstream facing ports.

Transfer One or more bus transactions to move information between a software client and its function.

Transfer Type Determines the characteristics of the data flow between a software client and its function. Four standard transfer types are defined: control, interrupt, bulk, and isochronous.

TS Thermal Shock

Turn-around Time The time a device needs to wait to begin transmitting a packet after a packet has been received to prevent collisions on the USB. This time is based on the length and propagation delay characteristics of the cable and the location of the transmitting device in relation to other devices on the USB.

UART Universal Asynchronous Receiver/Transmitter serial port

Universal Serial Bus Driver (USBD) The host resident software entity responsible for providing common services to clients that are manipulating one or more functions on one or more Host controllers.

Universal Serial Bus Resources Resources provided by the USB, such as bandwidth and power. See also **Device Resources** and **Host Resources**.

Upstream The direction of data flow towards the host. An upstream port is the port on a device electrically closest to the host that generates upstream data traffic from the hub. Upstream ports receive downstream data traffic.

USBD See **Universal Serial Bus Driver**.

USB-IF USB Implementers Forum, Inc. is a nonprofit corporation formed to facilitate the development of USB compliant products and promote the technology.

VBI Vertical Blanking Interval, also known as the “backporch”.

Virtual Device A device that is represented by a software interface layer. An example of a virtual device is a hard disk with its associated device driver and client software that makes it able to reproduce an audio.WAV file.

VLIO Variable Latency Input/Output interface.

YUV A method of characterizing video signals typically used in digital cameras and PAL television specifying luminance and chrominance.

WAP Wireless Application Protocol. WAP is a set of protocols that lets users of mobile phones and other digital wireless devices access Internet content, check voice mail and e-mail, receive text of faxes and conduct transactions. WAP works with multiple standards, including CDMA and GSM. Not all mobile devices support WAP.

W-CDMA Wideband CDMA, a third generation wireless technology under development that allows for high-speed, high-quality data transmission. Derived from CDMA, W-CDMA digitizes and transmits wireless data over a broad range of frequencies. It requires more bandwidth than CDMA, but offers faster transmission because it optimizes the use of multiple wireless signals, instead of one, as does CDMA.

Wireless LAN A wireless LAN uses radio frequency technology to transmit network messages through the air for relatively short distances, like across an office building or a college campus. A wireless LAN can serve as a replacement for, or an extension to, a traditional wired LAN.

Wireless MMX Intel® Wireless MMX™ technology integrates the high performance of Intel® MMX™ technology and the integer functions from Streaming SIMD Extensions (SSE) to the Intel® XScale™ microarchitecture. Intel® Wireless MMX™ technology brings the power of a 64-bit, parallel, multimedia architecture to handheld systems in a programmer's model that is already familiar to thousands of developers. Like Intel® MMX™ technology and SSE, Intel® Wireless MMX™ technology utilizes 64-bit wide Single Instruction Multiple Data (SIMD) instructions which allow it to process up to eight data elements in a single cycle. Refer to *The Complete Guide to Intel® Wireless MMX™ Technology*, order number 278626.

Wireless Spectrum A band of frequencies where wireless signals travel carrying voice and data information.

Word A data element that is four bytes (32 bits) in size.

WML Wireless Markup Language, a version of HDML is based on XML. Wireless applications developers use WML to re-target content for wireless devices.

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X