TILERA®

Tile Processor Architecture Overview for the TILE-Gx Series

Release 0.20
Doc. No. UG130
February 2012
Tilera Corporation

Copyright © 2012 Tilera Corporation. All rights reserved. Printed in the United States of America.

No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, except as may be expressly permitted by the applicable copyright statutes or in writing by the Publisher.

The following are registered trademarks of Tilera Corporation: Tilera and the Tilera logo.

The following are trademarks of Tilera Corporation: Embedding Multicore, The Multicore Company, Tile Processor, TILE Architecture, TILE64, TILE*Pro*, TILE*Pro36*, TILE*Pro64*, TILExpress, TILExpress-64, TILExpress*Pro*-64, TILExpress-20G, TILExpress*Pro*-20G, TILExpress*Pro*-22G, iMesh, TileDirect, TILEmpower, TILEmpower-Gx, TILEncore, TILEncore*Pro*, TILEncore-Gx, TILE-Gx, TILE-Gx9, TILE-Gx16, TILE-Gx36, TILE-Gx64, TILE-Gx100, TILE-Gx3000, TILE-Gx5000, TILE-Gx8000, DDC (Dynamic Distributed Cache), Multicore Development Environment, Gentle Slope Programming, iLib, TMC (Tilera Multicore Components), hardwall, Zero Overhead Linux (ZOL), MiCA (Multicore iMesh Coprocessing Accelerator), and mPIPE (multicore Programmable Intelligent Packet Engine). All other trademarks and/or registered trademarks are the property of their respective owners.

Third-party software: The Tilera IDE makes use of the BeanShell scripting library. Source code for the BeanShell library can be found at the BeanShell website (http://www.beanshell.org/developer.html).

This document contains advance information on Tilera products that are in development, sampling or initial production phases. This information and specifications contained herein are subject to change without notice at the discretion of Tilera Corporation.

No license, express or implied by estoppels or otherwise, to any intellectual property is granted by this document. Tilera disclaims any express or implied warranty relating to the sale and/or use of Tilera products, including liability or warranties relating to fitness for a particular purpose, merchantability or infringement of any patent, copyright or other intellectual property right.

Products described in this document are NOT intended for use in medical, life support, or other hazardous uses where malfunction could result in death or bodily injury.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS IS" BASIS. Tilera assumes no liability for damages arising directly or indirectly from any use of the information contained in this document.

Publishing Information:

Document number:	UG130
Release	0.20
Date	2/10/12

Contact Information:

Tilera Corporation
Information info@tilera.com **Web Site http://www.tilera.com**

Contents

CHAPTER 1 TILE-GX PROCESSOR OVERVIEW

1.1 TILE-Gx Processor 1

CHAPTER 2 TILE ARCHITECTURE

2.1 Processor Core 3
- 2.1.1 Instruction Set Architecture Overview 3
- 2.1.2 Processing Pipeline 4
- 2.1.3 Front End 7
  - 2.1.3.1 Instruction Cache 7
  - 2.1.3.2 Instruction TLB 7
  - 2.1.3.3 Instruction Prefetch 8
  - 2.1.3.4 Branch Prediction 8
- 2.1.4 Execution Units/Pipelines 8
- 2.1.5 Cache and Memory Architecture 8
  - 2.1.5.1 Tile Caches 10
  - 2.1.5.2 L1 DCache 10
  - 2.1.5.3 L2 Cache Subsystem 12
- 2.1.6 Special Purpose Registers (SPRs) 13
- 2.1.7 Interrupts and Exceptions 13
- 2.1.8 Protection Architecture 14
  - 2.1.8.1 Levels of Protection 14
  - 2.1.8.2 Protected Resources 14

2.2 Switch Interface and Mesh 16
- 2.2.1 The iMesh 16
- 2.2.2 Switch Interface 18
- 2.2.3 Switch Micro Architecture 19
  - 2.2.3.1 Arbitration 19
  - 2.2.3.2 Round Robin Arbitration 20
  - 2.2.3.3 Network Priority Arbitration 20
- 2.2.4 TILE-Gx Processor — Partitioning 21

CHAPTER 3 I/O DEVICE INTRODUCTION

3.1 Overview 23
- 3.1.1 Tile-to-Device Communication 23
- 3.1.2 Coherent Shared Memory 24
- 3.1.3 Device Protection 24
- 3.1.4 Interrupts 24

3.1.5 Device Discovery 25
3.1.6 Common Registers 25

CHAPTER 4 DDR3 MEMORY CONTROLLERS

4.1 Memory Striping 29
4.2 Rank/Bank Hashing 29
4.3 Memory Request Scheduling 29
4.4 Page Management Policy 29
4.5 Priority Control 30
4.6 Starvation Control 30
4.7 Performance Counters 30

CHAPTER 5 HARDWARE ACCELERATORS

5.1 Overview 31
5.1.1 Mesh Interface 32
5.1.2 TLB (Translation Lookaside Buffer) 32
5.1.3 Engine Scheduler 33
5.1.4 Function-Specific Engines 33
5.1.5 DMA Channels 33
5.1.6 PA-to-Header Generation 33
5.1.7 Operation 33
5.1.8 Crypto Accelerators 34
5.1.9 Compression Accelerators 35
5.1.10 MemCopy DMA Engine 35

CHAPTER 6 PCIE/TRIO

6.1 PCIe Interfaces 37

CHAPTER 7 XAUI/MPIPE

7.1 mPIPE Subsystem 39

CHAPTER 8 OTHER I/OS (USB, ETC.)

8.1 USB Subsystem 41
8.2 Flexible I/O System 41
8.3 UART System 41
8.4 I^2C Systems 42
8.5 SPI System 42

CHAPTER 9 DEBUG

CHAPTER 10 BOOT

GLOSSARY 47

INDEX 49

CHAPTER 1 TILE-GX PROCESSOR OVERVIEW

1.1 TILE-Gx Processor

This document provides an architectural overview of the TILE-Gx™ family of system-on-chip multicore processors, with an emphasis on the 36-core TILE-Gx8036. This document is not intended to provide an exhaustive description of the TILE-Gx family. For more information, please consult the *Instruction Set Architecture for TILE-Gx* (UG401), and *I/O Device Guide for the TILE-Gx Family of Processors* (UG404). TILE-Gx documentation can be found in the MDE document index.

The TILE-Gx family is fabricated in 40nm process technology, offering high clock speeds at very low power. With the standard part at 1.2GHz clock rate and with a typical networking application running, the device is expected to draw approximately 25-30W of power. Considering the high level of system integration (onboard memory controllers, PCI Express, and I/O controllers), the total system power is 2x to 4x lower than other competitive high performance processor systems.

The TILE-Gx8036 chip is packaged in a 37.5mm x 37.5mm flip-chip Ball Grid Array (BGA). It is RoHS-6 compliant and is compatible with standard multilayer PCB designs.

Each of the 36 processor cores is a full-fledged 64-bit processor with local cache and supports a flexible virtual memory system. Any of the cores can independently run its own operating system (that is, standard Linux 2.6), and many or all of the tiles can as a group run Symmetric Multi-Processing (SMP) operating systems, such as SMP Linux. Existing applications written in C or C++ for standard processors will port very quickly to the TILE-Gx™.

The development tools enable rapid migration to multiple tiles using various standard multicore programming techniques such as threads, decomposed pipelining, or multiple “run-to-completion” process instances.

Several unique architectural innovations enable the low-power and scalable performance of the Tile Processor™. The mesh-based interconnect between processor cores provides high communication bandwidth and low latency to other tiles cache, external memory and I/O. Further, the distributed and shared coherent cache architecture removes bottlenecks and contention and minimizes power dissipation.

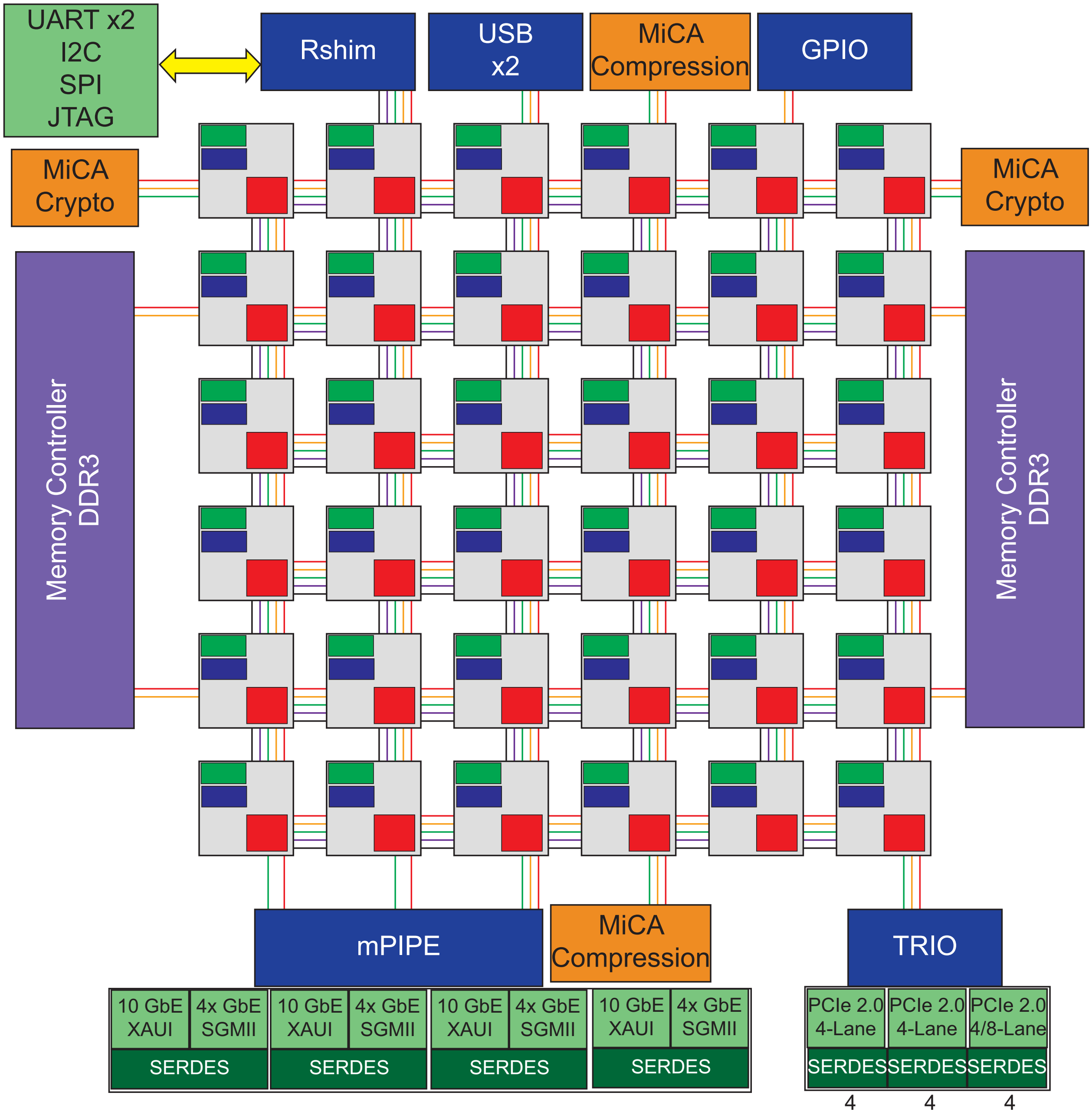
UART x2
I2C
SPI
JTAG
Rshim
USB
x2
MiCA
Compression
GPIO
MiCA
Crypto
MiCA
Crypto
Memory Controller
DDR3
Memory Controller
DDR3
mPIPE
MiCA
Compression
TRIO
10 GbE
XAUI
4x GbE
SGMII
SERDES
10 GbE
XAUI
4x GbE
SGMII
SERDES
10 GbE
XAUI
4x GbE
SGMII
SERDES
10 GbE
XAUI
4x GbE
SGMII
SERDES
PCIe 2.0
4-Lane
PCIe 2.0
4-Lane
PCIe 2.0
4/8-Lane
SERDES
SERDES
SERDES
4
4
4

Figure 1-1. TILE-Gx36 SoC Block Diagram

CHAPTER 2 TILE ARCHITECTURE

Each tile consists of a 64b in-order processor core (with its constituent pipelines, execution units, and L1 caches), an L2 cache subsystem, and a switch interface to the on-chip mesh interconnects.

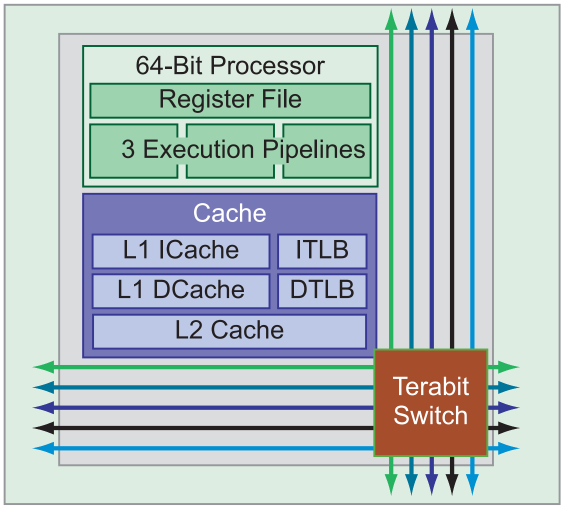

Figure 2-1. Detail of a Tile within the Tile Processor

2.1 Processor Core

The processor core is an in-order 3-wide 64-bit integer processing engine that uses a 64-bit VLIW (Very Long Instruction Word) architecture. Up to three instructions can be packed into a single instruction word. There are 64 architected registers, each 64b wide.

2.1.1 Instruction Set Architecture Overview

The Tile Processor instruction set architecture (ISA) includes a full complement of general-purpose RISC instructions and has been enhanced with a rich set of DSP and SIMD instructions for multimedia and signal processing applications. These include:

- SIMD instructions for sub-word parallelism
  - Eight 8-bit parallel, four 16-bit parallel, two 32-bit parallel
- Dot product
- Complex multiply
- Laned multiply (with accumulate)
- Laned compare
- Saturating arithmetic
- Min/Max/Average, Absolute Difference

- Conditional result support. In addition, some special instructions are included to accelerate various applications:
    - Byte shuffle
    - Bit field insert/extract (Signed, Unsigned)
    - Population count
    - Sum of absolute differences (SAD)
    - CRC for hashing and checksums
- Table lookup instructions
- Atomics
- Non temporal loads and stores

C intrinsics are provided in a Tilera-provided architecture-specific library for the enhanced instructions.

The instruction set includes a `nap` instruction that allows the programmer to idle tiles when they are not needed for computation. Using this feature conserves power.

At the Hypervisor and operating system levels, a large set of Special Purpose Registers (SPRs) provide access to architectural control and real-time processing states, an interrupt controller that can handle one interrupt per processing cycle on a priority basis (including those from I/O and external system components), and a timer that can generate time-based periodic interrupts.

The Tile Processor consists of four major sub-sections: the front end, execution units, L1 DCache, and L2 Cache. These sub-sections are described in the sections that follow.

2.1.2 Processing Pipeline

The TILE-Gx™ processor uses a short, in-order pipeline aimed at low branch and load-to-use latencies. The pipeline consists of six main stages: Fetch, Branch Predict, Decode, Execute0, Execute1, and WriteBack, as shown in Figure 2-2.

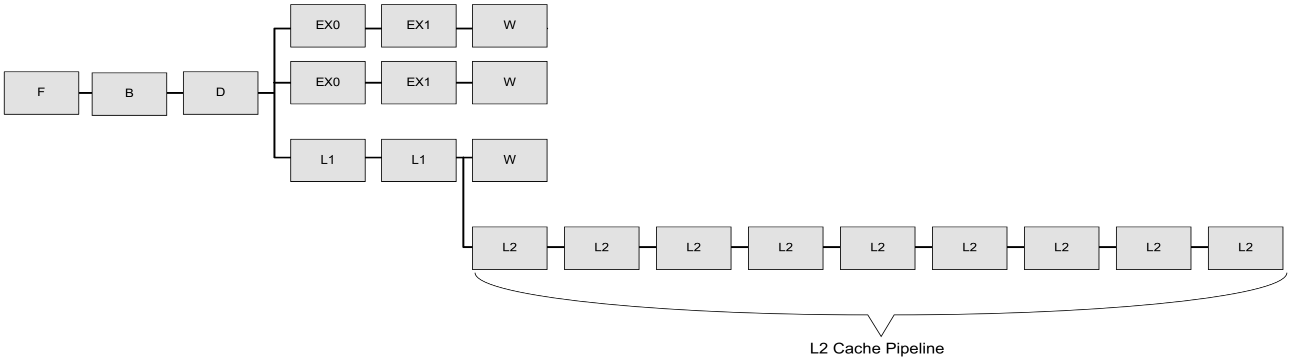

Figure 2-2. TILE-Gx Processor Pipeline

Fetch

During the Fetch pipeline stage the Program Counter (PC) is used to provide an index into the ICache and ITLB. The PC value can come from several sources including the next sequential instruction, branch target (including predicted taken branch), jump target, interrupt address, or boot address.

Branch Predict

During the Branch Predict pipeline stage predicted branch and jump instructions are decoded. For predicted taken branch instructions the pipeline is restarted at the branch target address. For `jrp` and `jalrp` instructions and the pipeline is restarted at the value on top of Return Address Stack.

Decode

During the Decode pipeline stage the Icache and ITLB lookups during Fetch stage are checked for hits.

Also, the execution resources required for the instruction are checked, including the source and destination operands. A three-instruction bundle can require up to seven source register operands and three destination register operands. Source operands can come from the general purpose register file or can be bypassed from earlier instructions for the case when the destination register has not yet been written into the register file.

An instruction can stall in Decode pipeline stage for reasons listed in Table 1.

Table 1. Pipeline Stalls

Stall Reason		Description
Source Operand RAW	ALU Destination	Source operand is a destination operand of a 2-cycle ALU instruction in previous instruction (1-cycle ALU instruction destination operands are bypassed so will not cause a stall). Note that this only covers GPRs in the register file (not network mapped GPRs).
	Load Destination	Source operand is destination of a load that is not complete. Note that this only covers GPRs in the register file (not network mapped GPRs).
	Network mapped GPR	Source operand is a network mapped GPR and no source data is available on that network. The latency is unbounded, however async interrupts can be taken during the stall.
ALU after Load WAW		An ALU instruction destination register is also the destination of an earlier load which missed L1 Dcache and is still pending. This includes network mapped GPRs.
Load after Load WAW		A load instruction destination register is also the destination of an earlier load which missed L1 Dcache and is still pending. This includes network mapped GPRs.
Network Mapped GPR destination		The destination of an instruction is a network mapped GPR, and the network buffer is full. This would normally only happen due to network congestion or a destination tile not consuming network data.
Accesses to Slow SPRs.		mfspr or mtspr to slow SPR
Memory Fence		mf instruction issued and previous loads or stores are not complete.
L2 Cache queue full		Memory instruction and L2 Cache input queue is full.
L1 DCache fill		Memory instruction when L1 Dcache is being filled.
DTLB Hazard		Memory instruction following mtspr to DTLB_INDEX SPR.
ICOH		icoh instruction. Stalls until any instruction prefetches in flight complete.
Nap		Nap instruction. Stalls until an async interrupt occurs.

Execute0

Execute0 pipeline stage is used for the execution of most ALU operations, for example, `add`, `sub` (subtract), `xor`, etc. The destination value of these operations will be bypassed if any instruction in Decode stage uses it as a source operand.

Conditional branches are resolved in this stage; if a branch was incorrectly predicted in Branch Predict stage, the pipeline will be restarted with the correct PC.

Execute1

Execute1 pipeline stage is used to complete two-cycle ALU operations, for example multiply. The destination value of these operations will be bypassed if any instruction in Decode stage uses it as a source operand.

Pending interrupts are evaluated in this stage; if an interrupt is to be taken the pipeline will be restarted with the appropriate PC for the highest priority interrupt.

Write Back

Destination operands are all written to the register file during Write Back pipeline stage. As mentioned earlier, source operands that are destinations that have not yet reached Write Back stage can be bypassed to following instructions as needed.

The destination operands of loads that miss the L1 Dcache are not written in Write Back stage; instead they are written to the register file when they return from memory (either L2 Cache, L3 Cache, or DRAM memory).

Pipeline Latencies

Table 2 shows the latencies for some important cases.

Table 2. Pipeline Latencies

Operation	Latency (Cycles)
Branch predict (correct)	0
Branch mispredict	2
Jump - use Return Address Stack (correct)	0
Jump - did not use Return Address Stack or used incorrect value	2
Access to slow SPR	3
Load to use - L1 Dcache hit	2
Load to use - L1 Dcache miss, L2 Cache hit	11
Load to use - L2 Cache miss, neighbor L3 Cache hit, Dword 0	32
Load to use - L2 Cache miss, neighbor L3 Cache hit, Dword 1 to 7	41
L1 Icache miss, prefetch hit	2
L1 Icache way mispredict	2
L1 Icache miss, L2 Cache hit	10
L1 Icache miss, L2 miss, neighbor L3 hit	40

2.1.3 Front End

The function of the processor front end is to control instruction processing. In order for an instruction to execute, several actions need to be completed. First, the instruction needs to be fetched from memory. The front end maintains the Icache, which stores local copies of recently used instructions to reduce memory latency. Next, the instruction needs to be decoded in order to determine what resources it uses, whether or not it alters the control flow, etc. After the instruction has been decoded, the processor needs to determine if the instruction is capable of executing. In order to do this, a dependency calculation is done to make sure that all of the appropriate operands and other resources are available. If not, the pipeline is stalled. Next, the input operands' data must be fetched from the appropriate location and supplied to the appropriate execution unit(s). Along the way, many conditions can arise such as interrupts, cache misses, and TLB misses. It is the responsibility of the front end to deal with these exceptional cases and appropriately steer the instruction stream.

2.1.3.1 Instruction Cache

The Level 1 Instruction Cache (abbreviated as L1 Icache or just Icache) provides instructions to the pipeline. The Icache is 32kB, 2-way associative. Table 3 lists some attributes of the Icache.

Table 3. ICache Attributes

Attribute	L1 ICache
Capacity	32 kB
Line Size	64 bytes
Lines	512
Associativity	2-way
Sets	256
Allocate Policy	Instruction fetch miss
Write Policy	N/A
Data Integrity	1 bit parity on Instruction 1 bit parity on Tag

2.1.3.2 Instruction TLB

The Icache is physically tagged and the program addresses are virtual, so in order to determine if the Icache lookup hits the virtual address must be translated to a corresponding physical address. The ITLB stores copies of virtual to physical translations. Table 4 lists the attributes of the ITLB.

Table 4. ITLB Attributes

Parameter	Value	Comment
Entries	16	
Associativity	16-way	Fully associative
Tag Compare	41 bits	30 VPN, 8 ASID, 2 CPL, 1 Valid
Tag Compare Control	5 bits	1 Global, 4 Page Size Mask
Data	43 bits	28 PFN, 8 LOTAR, 7 Status

2.1.3.3 Instruction Prefetch

Instruction prefetching is a trade off that reduces Icache miss latency (and thus stalls) by speculatively reading instructions that are likely (but not guaranteed) to be used, at the expense of higher memory bandwidth utilization.

I-Stream Prefetch Operation

Instruction stream prefetching is enabled by two settings in the `SBOX_CONFIG` SPR, which is a privileged SPR. When these settings are set they prevent prefetching from being used, unless privileged software permits it. One setting determines how many cache lines ahead (between 0 and 3) to deliver to the four-entry prefetch buffer. The second setting determines how many lines (between 0 and 3) to bring to the L2 Cache. Prefetching is triggered when an instruction fetch misses in the L1 Icache.

First, the number of cache lines specified for the prefetch buffer are requested, followed by the number of cache lines specified for the L2 Cache. After a miss, cache lines sequential to the line that missed are prefetched, based on the principle that those instructions are likely to be executed but are not in L1 Icache.

If there is a taken branch, jump, or interrupt prior to the requested cache lines coming back from memory, a new set of cache lines will be requested, with the newest lines being put into the prefetch buffer.

2.1.3.4 Branch Prediction

Branch prediction is responsible for early, speculative re-steering of the front end for conditional branches, offset jumps, and predicted indirect jumps. In all cases the correctness of the prediction is checked and, if not correct, the front end is re-steered again to the correct flow.

For a correctly predicted branch, no instruction slots are lost; for an incorrectly predicted branch, two instructions slots are lost.

Return Address Stack

`jrp` and `jalrp` instructions have a hint to use a Return Address Stack (RAS) as the target. The hint to put the return value onto the RAS is given in instructions `jal`, `jalr`, and `jalrp`. The address at the top of RAS is used similar to predicted conditional branches; for a correct return address found on RAS no instruction slots are lost; if not, two instructions slots are lost.

The depth of RAS is four entries.

2.1.4 Execution Units/Pipelines

The TILE-Gx Processor Engine has three execution pipelines (P0, P1, P2). P0 is capable of executing all arithmetic and logical operations, bit and byte manipulation, selects, and all multiply and fused multiply instructions. P1 can execute all of the arithmetic and logical operations, SPR reads and writes, conditional branches, and jumps. P2 can service memory operations only, including loads, stores, and atomic memory access instructions.

2.1.5 Cache and Memory Architecture

The Tile Processor architecture defines a flat, globally shared 64-bit physical address space and a 64-bit virtual address space (note that Tile-Gx processors implement a 40-bit subset physical address and 42-bit subset virtual address). Memory is byte-addressable and can be addressed in 1, 2, 4 or 8 byte units, depending on alignment. Memory transactions to and from a tile occur via the iMesh.

The globally shared physical address space provides the mechanism by which software running on different tiles, and I/O devices, share instructions and data. Memory is stored in off-chip DDR3 DRAM.

Page tables are used to translate virtual addresses to physical addresses (page size range is 4 kB to 64 GB). The translation process includes a verification of protected regions of memory, and also a designation of each page of physical addresses as either coherent, non-coherent, uncacheable, or memory mapped I/O (MMIO). For coherent and non-coherent pages, values from recently-accessed memory addresses are stored in caches located in each tile. Uncacheable and MMIO addresses are never put into a tile cache.

The Address Space Identifier (ASID) is used for managing multiple active address spaces. Recently-used page table entries are cached in TLBs (Translation Lookaside Buffers) in both tiles and I/O devices.

Hardware provides a cache-coherent view of memory to applications. That is, a read by a tile or I/O device to a given physical address will return the value of the most recent write to that address, even if it is in a tile's cache. Instruction memory that is written by software (self-modifying code) is *not* kept coherent by hardware. Rather, special software sequences using the `icoh` instruction must be used to enforce coherence between data and instruction memory.

Cache management instructions include cache flush and invalidate instructions. Atomic operations include `FetchAdd`, `CmpXchg`, `FetchAddGez`, `Xchg`, `FetchOr`, and `FetchAnd`. Memory ordering is relaxed, and a memory fence instruction provides ordering between cores when needed.

Illegal Virtual Address

The virtual address is architecturally 64 bits, but is implemented as 42 bits in the Tile-Gx processor. Virtual addresses that are not sign-extended values (i.e. bits[63:41] of the VA are all 0's or all 1's) are illegal — the implication of this is that there are two legal VA regions, lower and upper, and an illegal region in the middle, as shown in Table 5.

.

Table 5. Virtual Address Space

Address	Region
$2^{64}-1$	**Upper VA Region**
...	
$2^{64}-2^{41}$	
$2^{64}-2^{41}-1$	**Illegal VA Region**
...	
2^{41}	
$2^{41}-1$	**Lower VA Region**
...	
0	

It is illegal to do a memory operation (for example `load` or `store`), or to execute instructions from an illegal VA, or to take a branch from the lower to upper VA region (or vice-versa). An attempt to do so will result in an exception.

2.1.5.1 Tile Caches

Figure 2-3 shows a high-level block diagram showing how the caches relate to the processor and switch.

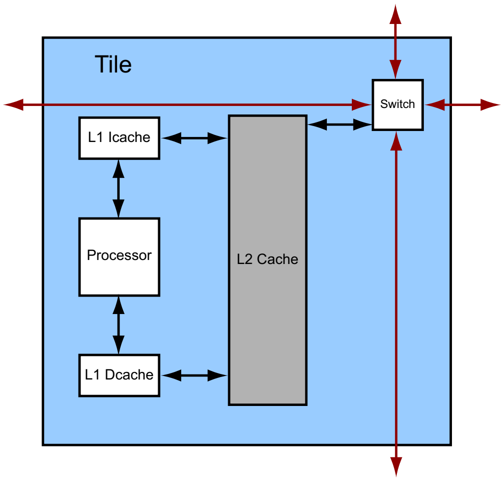

Figure 2-3. Tile Block Diagram Showing L2 Cache

The cache subsystem is non-blocking and supports multiple concurrent outstanding memory operations; it supports hit under miss and miss under miss, allowing loads and stores to different addresses to be re-ordered to achieve high bandwidth and overlap miss latencies, while still ensuring that true memory dependencies are enforced. A memory fence instruction (`mf`) can be used to enforce memory ordering when required.

The Tile Processor does not stall on load or store cache misses. Rather, execution of subsequent instructions continues until the data requested by the cache miss is actually needed by the consuming instruction. The cache system maintains coherence with I/O accesses to memory, and allows I/O devices to read and write the caches directly.

In addition to normal loads and stores, the tile architecture provides atomic memory operations (`exch`, `cmpexch`, `fetchadd`, `fetchaddgez`, `fetchand`, `fetchor` instructions).

2.1.5.2 L1 DCache

The L1 Data Cache (abbreviated as L1 DCache) stores copies of data from recently accessed addresses. Table 6 shows some attributes of the L1 DCache.

Table 6. L1 DCache Attributes

Attribute	L1 Data Cache
Capacity	32 kB
Line Size	64 bytes
Lines	512
Associativity	2-way
Sets	256

Table 6. L1 DCache Attributes (continued)

Attribute	L1 Data Cache
Allocate Policy	Load miss
Write Policy	Write through
Data Integrity	Byte parity on Data, 1 bit parity on Tag

The L1 DCache is physically tagged. A Data TLB (DTLB) is used to translate virtual addresses to the corresponding physical addresses. The DTLB stores copies of virtual to physical translations for data stream accesses. Table 7 lists the attributes of the DTLB.

Table 7. Data TLB Attributes

Parameter	Value	Comment
Entries	32	
Associativity	32	Fully associative
Tag Compare	41 bits	30 VPN, 8 ASID, 2 CPL, 1 Valid
Tag Compare Control	5 bits	1 Global, 4 Page Size
Data	43 bits	28 PFN, 8 LOTAR, 7 Status

The Dcache is non-blocking. For DSP applications, non-temporal streaming loads and stores are supported for optimized cache usage.

D-Stream Prefetch

A configurable data-stream prefetch engine provides a significant performance boost without requiring software prefetching. Once enabled, it autonomously prefetches data from memory. This provides an alternative approach to placing prefetch instructions in the instruction stream. However, both approaches can be used concurrently, for example explicit prefetch instructions could be used to prefetch data from addresses that the prefetch engine would otherwise miss.

The D-Stream Prefetch operation is enabled by setting a bit in `SBOX_CONFIG` SPR. This is a privileged SPR; this protects against prefetching being used unless privileged software complies. Once prefetching is enabled, user software can control the other parameters in `DSTREAM_PF` SPR, which is a non-privileged SPR. The parameter are listed in Table 8.

Table 8. D-Stream Prefetch Parameters in DSTREAM_PF SPR

Name	Description
Stride	This value is added to the load address to determine the address to prefetch. A value of 1 would prefetch the next successive cache line, a value of 2 the cacheline 2 away, etc. A value of 0 indicates no prefetch.
Level	Which level of cache to prefetch to: L1 Dcache, L2 Cache, or L3 Cache.
Miss_Only	Indicates whether to arm the prefetcher on any load, or only loads that miss L1 Dcache.

The prefetching operation is initiated when a load instruction is executed, which conditionally records the address and arms the prefetcher (for example it will not be armed if prefetching is setup to only happen on a L1 Dcache miss and the load hits). Once armed, the prefetcher sends a prefetch request to the L1 Dcache. If it hits in L1 Dcache, then the operation is complete; if it misses then the L2 Cache is checked. If it misses in the L2 Cache then a read request is sent to memory. The prefetched data is placed in the appropriate cache as defined in the Level parameter.

2.1.5.3 L2 Cache Subsystem

Table 9 lists some characteristics of the L2 Cache.

Table 9. L2 Cache Attributes

Attribute	L2 Cache
Capacity	256 kB
Line Size	64 bytes
Lines	4096
Associativity	8-way
Sets	512
Allocate Policy	Load or store miss (home tile); Load (non-home tile)
Write Policy	Write back
Data Integrity	ECC (single bit correct, double bit detect) on Data 1 bit parity on Tag

Cache Operation

The processor can issue one load or one store per cycle. The L1 Dcache is checked for the requested data. If it does not have the requested data, the request is sent to the L2 Cache. Stores update the L1 Dcache if the targeted cache block is present, and always write through to the L2 Cache. If the L2 Cache does not have the requested data, then a request is sent to the home tile (if the tile is not the home tile) or to the DRAM controller (if the tile is the home tile). If the home tile's L2 Cache does not have the requested cache line, it in turn sends a request to the DRAM controller and delivers the data to the requesting tile.

Instruction fetches that miss in the L1 Icache are sent to L2 Cache, which then handles them in the same way as L1 Dcache misses described in the previous paragraph.

I/O devices also send memory accesses to the home tile. The request will be completed immediately if the address is found in L2 Cache, otherwise the tile will send a request to the DRAM controller.

The L2 cache subsystem supports up to 8 outstanding cacheline requests to DDR memory and/or to other caches. The L2 subsystem contains an 8-entry (64B/entry) coalescing write buffer that merges writes to the same cacheline before writing through to the home tile, saving bandwidth and power consumption on the network.

Dynamic Distributed Caching

TILE-Gx uses Dynamic Distributed Caching (DDC) to provide a hardware-managed, cache-coherent approach to shared memory. Each address in physical memory space is assigned to a home tile. The mechanism for assigning the home is flexible; for example a specific tile can be chosen, or addresses can be distributed across many tiles. Data from any address can be cached at the home tile and also remotely by other tiles. This mechanism allows each tile to view the collection of all tiles' caches as a large shared, distributed coherent cache. It promotes on-chip access and avoids the bottleneck of off-chip global memory.

2.1.6 Special Purpose Registers (SPRs)

The Tile Processor contains special purpose registers (SPRs) that are used for several reasons:

- Hold state information and provide locations for storing data that is not in the general purpose register file or memory
- Provide access to structures such as TLBs
- Control and monitor interrupts
- Configure and monitor hardware features, for example prefetching, iMesh routing options, etc.

SPRs can be read and written by tile software (via `mfspr` and `mtspr` instructions, respectively), and in some cases are updated by hardware.

The SPRs are grouped by function into protection domains, each of which can be set to an access protection level, called the minimum protection level (MPL) for that protection domain. The "Protection Architecture" on page 14 defines how the MPLs are used.

2.1.7 Interrupts and Exceptions

Interrupt and exceptions are conditions that cause an unexpected change in control flow of the currently executing code. Interrupts are asynchronous to the program; exceptions are caused directly by execution of an instruction.

Some examples of interrupts are:

- UDN data available
- Tile timer interval expired
- Inter-processor interrupt (IPI), which could be result of:
  - I/O device operation completed
  - Packet available on the mPIPE network interface
  - IPI sent from a different tile

Some examples of exceptions are:

- Attempt to execute a privileged instruction from a non-privileged program
- `load` or `store` instruction to a virtual address that is not in DTLB
- Instruction fetch to a virtual address that is not in ITLB
- Software interrupt instruction (`swint`)

Interrupts and exceptions are tile specific, meaning that they are only reported to the local tile to which they are relevant. By localizing the reporting, no global structures or communication are needed to process the interrupt or exception. If action is required of a remote tile, it is software responsibility to communicate that need via one of the inter-tile communication mechanisms.

The interrupt/exception structure of the tile architecture is tightly integrated with the protection model, discussed in the next section. Each interrupt/exception type provides a programmable Minimum Protection Level (MPL) at which the corresponding handler executes.

On an interrupt or exception, the PC is loaded with an address derived from the `INTERRUPT_VECTOR_BASE` SPR, the specific type of interrupt or exception, and the protection level at which the handler will execute. This minimizes latency by dispatching directly to the appropriate handler at the appropriate protection level. The PC at which the interrupt or exception occurred is automatically saved to allow the handler to resume the interrupted flow.

2.1.8 Protection Architecture

This section discusses tile protection levels, also referred to as access or privilege levels.

2.1.8.1 Levels of Protection

The Tile architecture contains four levels of protection. The protection levels are a strict hierarchy, thus code executing at one protection level is afforded all of the privileges of that protection level and all lower protection levels. The protection levels are numbered 0-3 with protection level 0 being the least privileged protection level and 3 being the most privileged protection level. Table 10 presents one example use of the protection level number to names; other protection schemes different from the example are possible.

Table 10. Example Protection Level Usage

Protection Level	Usage
0	User
1	Supervisor
2	Hypervisor
3	Virtual Machine Level

2.1.8.2 Protected Resources

The Tile architecture contains several categories of protection mechanisms. These include:

- Prevention of illegal instruction execution
- Memory protection via multiple translation lookaside buffers (TLBs)
- Instructions to prevent the access selected networks
- Negotiated Application Programmer Interfaces (APIs) for physical device multiplexing
- Control of the protection level of an interrupt or exception

More description for each of the above is provided below.

Preventing Illegal Instruction Execution

The most basic protection mechanism prevents the execution of certain instructions unless a sufficient protection level is achieved. Not all instructions should be available to all programs, because some instructions are capable of modifying machine state beyond the scope of what a restricted process should be capable of modifying. Thus, the instruction set is partitioned into non-privileged and privileged instructions. The Tile architecture has a very small set of privileged instructions, namely `mtspr`, `mfspr`, and `iret`.

Also, some instruction encodings are invalid or architecturally reserved. Detecting invalid and reserved instructions is useful for finding bugs in a program, and allows an architecture to be extended via software emulation of ISA (Instruction Set Architecture) extensions.

An attempt to execute an instruction that is either beyond the program's protection level, or illegal or reserved, will result in a exception.

Memory Protection

A translation lookaside buffer (TLB) is primarily utilized to translate a virtual address to physical address on memory access. In the Tile architecture, the TLB is also used to store protection information in addition to translation information. The translation information also provides a form of protection by only allowing access to a subset of the address space. An attempt to access memory outside that subset will result in a exception. Additional protection information can also mark a page as read-only or as a member of a certain eviction class.

Preventing Instructions from Accessing Selected Networks

The Tile architecture both allows or disallows execution of instructions that access a particular network. By using this feature, access to UDN and/or IDN can be restricted to specific processes.

Negotiated APIs for Physical Device Multiplexing

One problem that arises in a system where multiple entities communicate with the same input/output devices involves device multiplexing. Multiple software processes on different tiles need to interact with one physical hardware device, for instance a network device, at the same time. To make this problem more challenging, the multiple software drivers must be protected from one another. To solve this problem, the Tile architecture utilizes virtual device APIs to allow physical device multiplexing. With this approach, the physical device driver resides in a hypervisor layer. The system level driver uses a virtualized device API to interface with the hypervisor. It is the hypervisor's responsibility to verify that the desired operation is legitimate and to provide a way to multiplex the physical device between multiple supervisor drivers.

Controlling the Protection Level of an Interrupt or Exception

The Tile Processor provides a mechanism to restrict access to the registers controlling protection levels. One function these registers control is the protection level. They specify what protection level is used when an interrupt or exception is taken. This mechanism is critical to building a protected machine and is described in the next section. When a process attempts to change a protection level, but does not have the appropriate privilege level needed, it causes a general protection violation interrupt.

Protection Levels

Every portion of the Tile architecture that requires protection has an associated MPL (Minimum Protection Level) register. This register contains a value from 0-3. The protection level contained in the MPL register specifies the protection level needed to complete a desired operation without faulting. Each tile contains a CPL (Current Protection Level) register that determines the privilege level of that tile.

When a protected event or action is initiated, the MPL required for the action is compared with the CPL of the tile. If the MPL is less than or equal to the CPL, then the action completes. On the other hand, if the MPL is greater than the CPL, an exception occurs (the exception is sent before the violating action completes). The interrupt handler associated with the specific violation is invoked. The processor's CPL is set to the minimum of the interrupt's minimum protection level and the current CPL (that is, if the tile's CPL is already greater than or equal to the MPL associated with the exception, the CPL is not changed, otherwise it is raised to the MPL).

The Tile architecture supports many MPL registers, which are mapped into the processor's SPR (Special Purpose Register) space. Some strict rules must be followed in order to set MPL registers. A process executing with a CPL at or above the MPL contained in a MPL register is capable of changing the MPL register arbitrarily lower or higher up to the process's CPL without faulting. If a process attempts to change a MPL register that has a higher value than its CPL, or attempts to set a MPL register to a value higher than its CPL, a general protection violation interrupt occurs.

MPL registers are prepended with the MPL acronym, for example the Minimum Protection Level for Boot Access register is named `MPL_BOOT_ACCESS`.

How to Set Protection Levels

Each MPL contains a read-only SPR named `MPL_`*`protected_resource`*, which is protected by the MPL itself. In order to set the MPL, a write is done to the `MPL_`*`protected_resource`*`_SET_n` register.

where:

`protected_resource` is the resource class being protected.

`n` is the level to which it is set.

Writing to these registers requires a CPL that has a maximum value of the `MPL_protected_resource` register and the level to which the MPL is being set (*n* above). This requirement guarantees that an MPL register can only be set to a value below the CPL of the executing process.

For example, to set `MPL_ITLB_MISS` to a protection level of 2 the system software needs to write a 1 to the `MPL_ITLB_MISS_SET_2` SPR, which sets the `MPL_ITLB_MISS` SPR to the value to 2.

2.2 Switch Interface and Mesh

2.2.1 The iMesh

All communication within the tile array and between the tiles and I/O devices takes place over the iMesh™ Interconnect, shown in Figure 2-4. The iMesh Interconnect consists of two classes of networks, both supporting low latency and high bandwidth communication. The first class comprises a set of software visible networks for application level streaming and messaging, while the second consists of the networks used by the memory system to handle memory requests, exchange cache coherency commands and support high performance shared memory communication. Dedicated switches are used to implement the iMesh Interconnect, allowing for a complete decoupling of data routing from the processor.

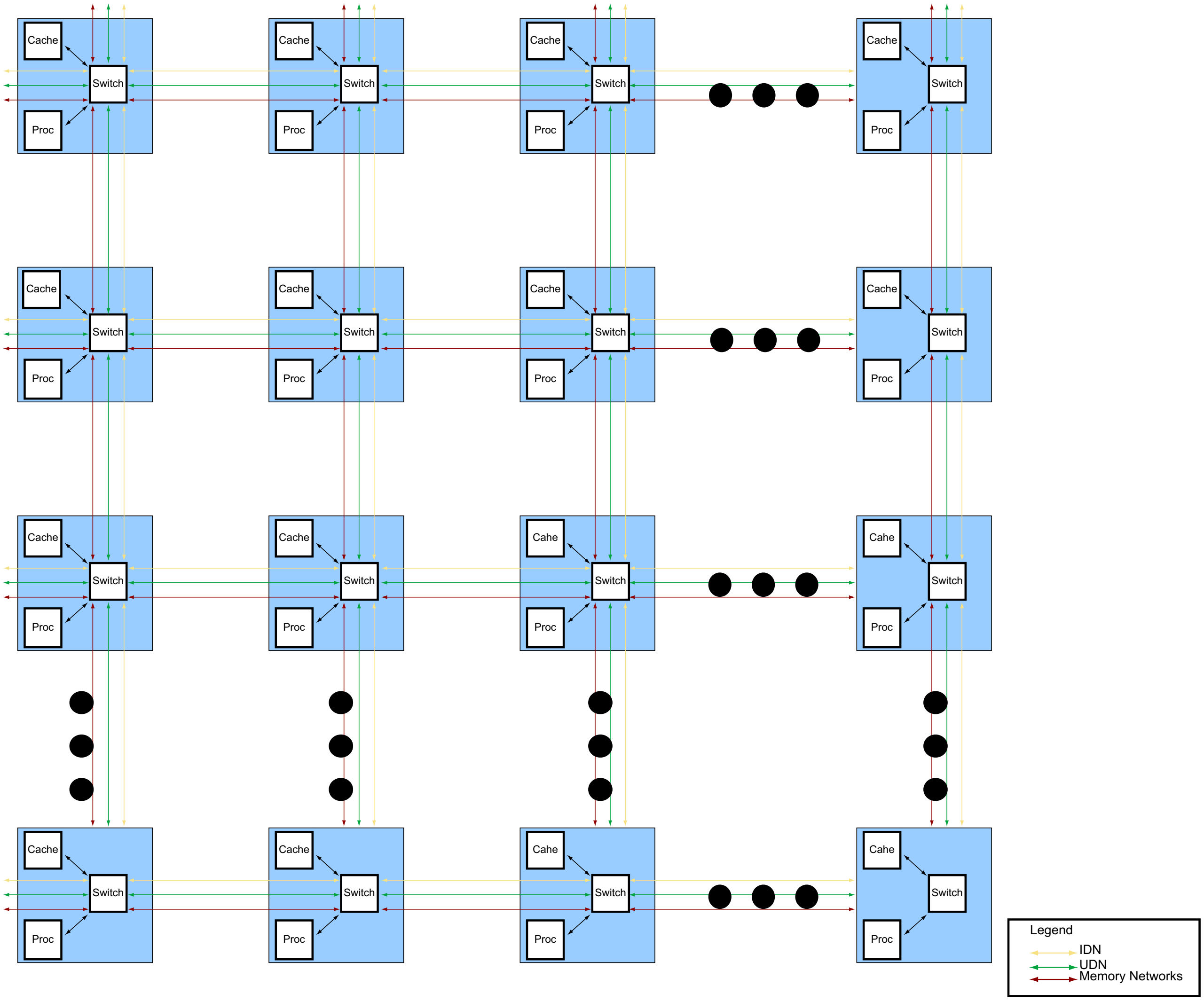
Cache
Switch
Proc
Cahe
Legend
IDN
UDN
Memory Networks

Figure 2-4. TILE-Gx Switch Interfaces

2.2.2 Switch Interface

Each switch point is a crossbar, and each switch connects for neighboring switches (N, S, E, and W) with the processor core, as shown in Figure 2-5. When input passes from one switch to the input of a neighboring switch it takes a single cycle (latency).

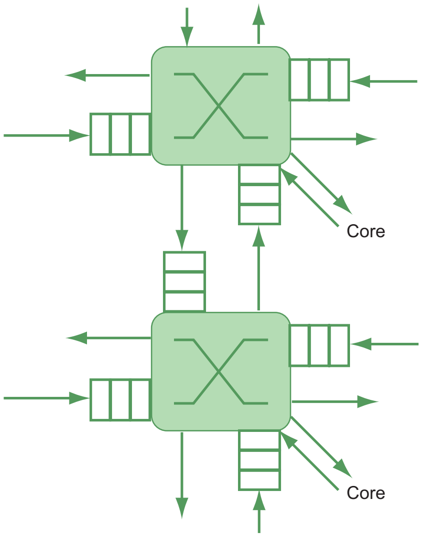

Figure 2-5. Switch Points

A "packet" is a message on the network (for example, a cache line from memory to a tile's cache), divided into units the width of the network (flits), plus a header. The header is used to specify the destination for the packet, illustrated in Figure 2-6.

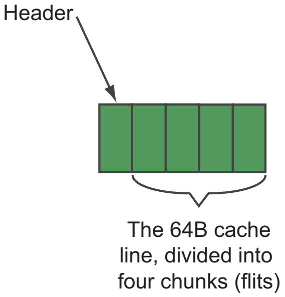

Figure 2-6. Header Configuration

The header arbitrates for an output port. Once the output port is granted, the packet is routed at the rate of one flit per cycle. Each output port maintains a "credit" counter for how many flit-sized entries are available in the connecting switch input buffer, and will route flits until the credit count is zero. Other packets requiring the same output port are blocked until the current packet has finished routing. This is often called "wormhole routing". As shown in Figure 2-7, the packet from West to North will be sent after the current packet from East to North has been completed sent.

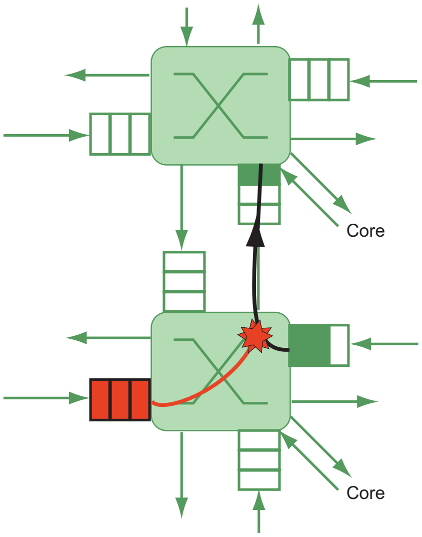

Figure 2-7. Packet Processing

2.2.3 Switch Micro Architecture

2.2.3.1 Arbitration

When multiple input ports require the same output port, the arbitration logic, shown as the Arbiter in Figure 2-8, must determine which input port is granted use of the output port.

The TILE-Gx devices support two flavors of arbitration, round robin arbitration (RRA), and Network Priority Arbitration (NPA).

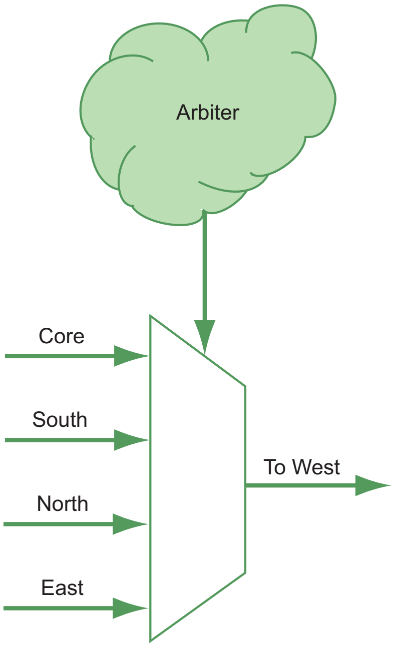

Figure 2-8. Arbitration Mechanism

2.2.3.2 Round Robin Arbitration

When multiple input ports want the same output port (see Figure 2-9), the following guidelines are followed:

- Give each input port equal priority.
- Process all requests in a round robin fashion through the input ports, which need the required output port.
- When nobody wants an output port, reset to a default value.
- Act locally fair, which treats requesting ports globally unfairly.

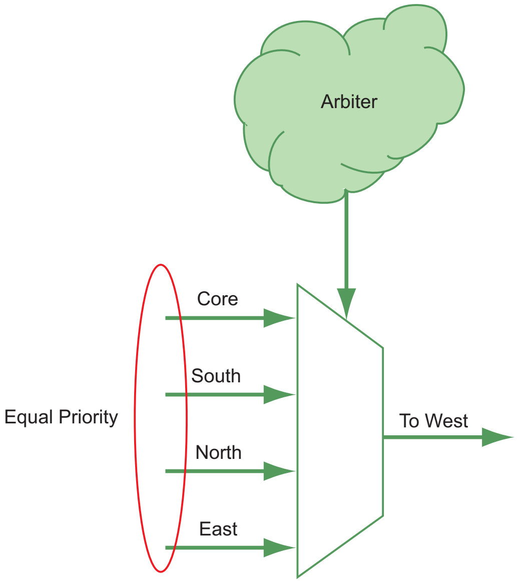

Figure 2-9. Round Robin Arbitration

2.2.3.3 Network Priority Arbitration

Give priority to packets already in the network (see Figure 2-10) as follows:

- Give Round Robin arbitration to network inputs.
- Only grant access to requests from the Core when no network inputs need the output port.
- Provide a configurable starvation counter for the core.
- Prevents cores on the edge from impacting high speed I/Os.
- Act locally fair, which treats requesting ports globally unfairly.

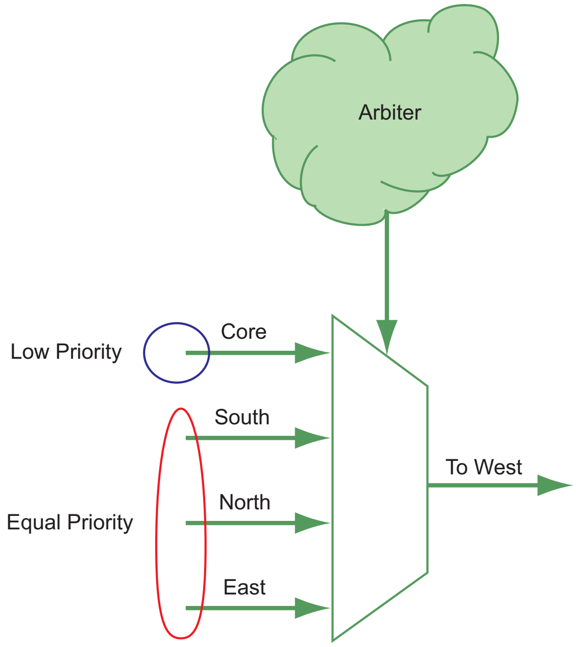
Arbiter
Low Priority
Core
South
Equal Priority
North
East
To West

Figure 2-10. Network Priority Arbitration

2.2.4 TILE-Gx Processor — Partitioning

The tile array within the chip can be partitioned to create virtualized computing “islands” with any number of tiles ranging from 1 to 36. This partitioning is enabled with software at the Hypervisor level and is enforced via standard TLB based memory partitioning as well as Tilera’s patented Hardwall™ protection. Refer to Figure 2-11 for an example of how a TILE-Gx processor can be partitioned.

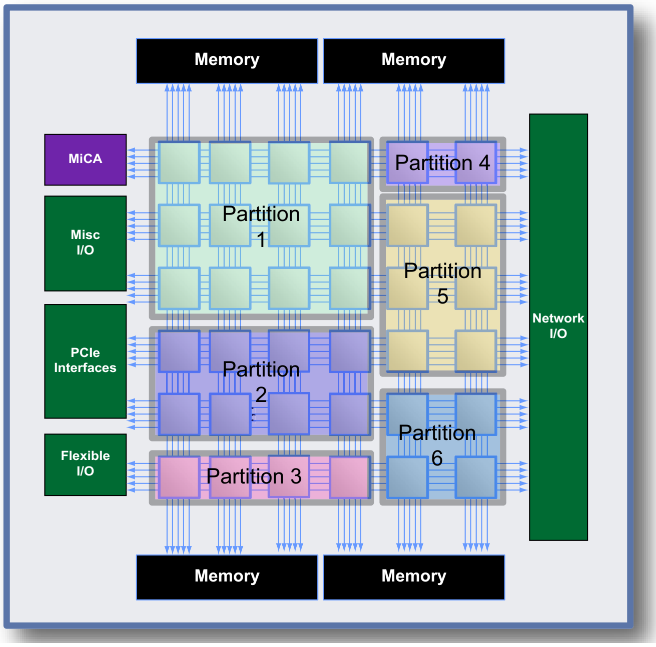
Memory
Memory
MiCA
Misc I/O
PCIe Interfaces
Flexible I/O
Partition 1
Partition 2
Partition 3
Partition 4
Partition 5
Partition 6
Network I/O
Memory
Memory

Figure 2-11. TILE-Gx Block Diagram — with Partitioning

Other system approaches include utilizing the cores as a unified “sea of computing” power where any tile can be called upon for application processing, often in a “run to completion” mode. The nature of the Tile Architecture makes it straightforward to explore different parallel programming models to select the best one for the system requirements.

CHAPTER 3 I/O DEVICE INTRODUCTION

3.1 Overview

The TILE-Gx™ family of processors contains numerous on-chip I/O devices to facilitate direct connections to DDR3 memory, Ethernet, PCI Express, USB, I^2C, and other standard interfaces.

This chapter provides a brief overview of the on-chip I/O devices. For additional information about the I/O devices, refer to *Tile Processor I/O Device Guide* (UG404). For detailed system programming information refer to the Special Purpose Registers (SPRs) and the associated device API guides.

3.1.1 Tile-to-Device Communication

Tile processors communicate with I/O devices via loads and stores to MMIO (Memory Mapped IO) space. The page table entries installed in a Tile's Translation Lookaside Buffer (TLB) contain a `MEMORY_ATTRIBUTE` field, which is set to MMIO for pages that are used for I/O device communication.

The X,Y fields in the page table entry indicate the location of the I/O device on the mesh and the translated physical address is used by the I/O device to determine the service or register being accessed.

Since each I/O TLB entry contains the X,Y coordinate of the I/O device being accessed, each device effectively has its own 40-bit physical address space for MMIO communication that is not shared with other devices or Tile physical memory space.

This physical memory space is divided into the fields shown in Figure 3-1 and defined in Table 11.

Note: Not all I/O devices use this partitioning. For example MICA does not have Regions or Service Domains. It uses a different type of division, which is described in Chapter 10 of the *Tile Processor I/O Device Guide* (UG404).

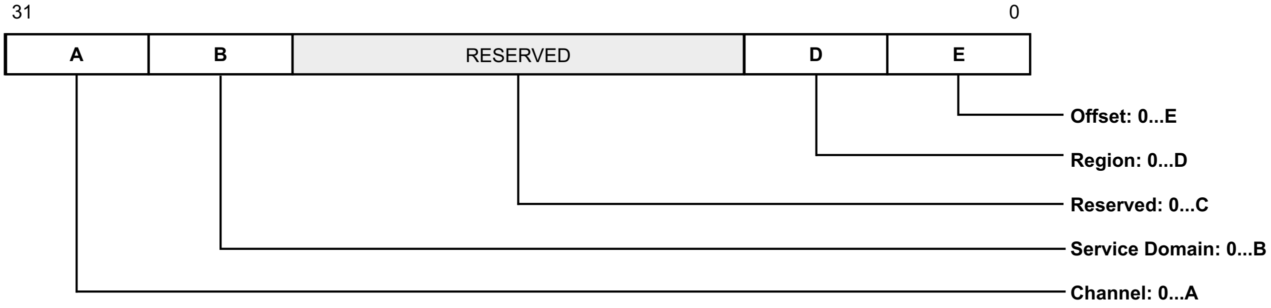

Figure 3-1. TILE-Gx Device Address Space

Table 11. TILE-Gx Physical Memory Space Descriptions

Bits	Bit Name	Required	Size	Description
A	Channel	No	Variable	Used when more than one device shares the same mesh location.
B	Service Domain	No	Variable	Used to index “permissions” table and allow/deny access to specific device services.
C	Reserved	No	Variable	Any “middle” bits of address that are not used.
D	Region	No	Variable	Selects service being accessed (for example register space vs. DMA descriptor post).
E	Offset	Yes	Variable	Address within the “Region” being accessed.

Each device has registers in Region-0 used to control and monitor the device. Devices can also implement additional MMIO address spaces for device communication protocols, such as posting DMA descriptors or returning buffers. System software is responsible for creating and maintaining the page table mappings that provide access to device services.

3.1.2 Coherent Shared Memory

I/O devices that provide bulk data transport utilize the high-performance, shared memory system implemented on TILE-Gx processors. All Tile memory system reads and writes initiated from an I/O device are delivered to a *home* Tile as specified in the physical memory attributes for the associated cacheline.

I/O TLBs and/or memory management units (MMUs) are used to translate user or external I/O domain addresses into Tile physical addresses. This provides protection, isolation, and virtualization via a standard virtual memory model.

3.1.3 Device Protection

In addition to the protection provided by the TLB for MMIO loads and stores, devices can provide additional protection mechanisms via the service domain field of the physical address. This allows, for example, portions of a large I/O physical address space to be fragmented, such that services can be allowed/denied to particular user processes without requiring dedicated (smaller) TLB mappings for each allowed service.

3.1.4 Interrupts

Devices interrupts are delivered to Tile software via the Tile Interprocess Interrupt (IPI) mechanism. Each Tile has four IPI MPLs, each with 32 interrupt events. I/O interrupts have programmable bindings in their MMIO register space, which specify the target Tile, interrupt number (also referred to as the IPI Minimum Protection Level or IPI MPL), and event number.

System software can choose to share Tile interrupt event bits among multiple I/O devices or dedicate the interrupt bits to a single I/O interrupt. Interrupt bits can also be shared between I/O and Tile-to-Tile interrupts.

I/O devices implement interrupt status and enable bits to allow interrupt sharing and coalescing.

3.1.5 Device Discovery

To facilitate a common device initialization framework, the TILE-Gx processors contain registers and I/O structures that allow non-device-specific software to "discover" the connected I/O devices for a given chip. After discovery, device-specific software drivers can be launched as needed.

All TILE-Gx processors contain an Rshim. The Rshim contains chip-wide services including boot controls, diagnostics, clock controls, reset controls, and global device information.

The Rshim's RSH_FABRIC_DIM, RSH_FABRIC_CONN, and RSH_IPI_LOC registers provide Tile-fabric sizing, I/O connectivity, and IPI information to allow software to enumerate the various devices. The common registers located on each device contain the device identifier used to launch device-specific driver software.

In order for Level-1 boot software to perform discovery, it must first find the Rshim. This is done by reading the `RSH_COORD` SPR located in each Tile.

Thus the basic device discovery flow is:

1. Read the RSHIM_COORD SPR to determine the Rshim location on the mesh.
2. Install an MMIO TLB entry for the Rshim.
3. Read the RSH_FABRIC_CONN vectors from Rshim to determine I/O device locations.
4. Install MMIO TLB entries for each I/O device.
5. Read the RSH_DEV_INFO register from each device to determine what the device type is, and launch any device-specific software.

3.1.6 Common Registers

While each device has unique performance and API requirements, a common device architecture allows a modular software driver model and device initialization process. The first 256 bytes of MMIO space contains the "common" registers that all I/O devices implement.[1] The common registers are used for device discovery as well as basic physical memory initialization and MMIO page sizing.

Table 12. Common Registers

Register	Address	Description
DEV_INFO	0x0000	This provides general information about the device attached to this port and channel.
DEV_CTL	0x0008	This provides general device control.
MMIO_INFO	0x0010	This provides information about how the physical address is interpreted by the I/O device.
MEM_INFO	0x0018	This provides information about memory setup required for this device.
SCRATCHPAD	0x0020	This is for general software use and is not used by the I/O shim hardware for any purpose.
SEMAPHORE0 and SEMAPHORE1	0x0028 and 0x0030	This is for general software use and is not used by the I/O shim hardware for any purpose.

---

[1] The "common registers" are located from 0x0000-0x0058.

Table 12. Common Registers (Cont'd)

Register	Address	Description
CLOCK_COUNT	0x0038	This is for general software use and is not used by the I/O shim hardware for any purpose.
HFH_INIT_CTL	0x0050	Initialization control for the hash-for-home tables.
HFH_INIT_DAT	0x0058	Read/Write data for hash-for-home tables.

Each of the major register sets (for example: the GPIO, UART, and MICA Crypto registers) for a specific device includes the common registers in the register set. The `SCRATCHPAD` register, for example, is a common register included in each of the register sets. The register set name prepends the register name as follows:

GPIO Register:	`GPIO_SCRATCHPAD` register
UART Register:	`UART_SCRATCHPAD` register
MICA_CRYPTO Register:	`MICA_CRYPTO_SCRATCHPAD` register

Registers beyond 0x100 contain the device specific registers.

Register definitions can be found as part of the MDE build and are located in the `HTML` directory. The directory structure is as follows:

- Memory Controller
- GPIO
- Rshim
- I^{2}C Slave
- I^{2}C Master
- SROM
- UART

Compression

- MiCA Compression Global
- MiCA Compression Inflate Engine
- MiCA Compression Deflate Engine
- MiCA Compression User Context
- MiCA Compression System Context

Crypto

- MiCA Crypto Global
- MiCA Crypto Engine
- MiCA Crypto User Context
- MiCA Crypto System Context

MPIPE / MACs

- mPIPE
- XAUI (Interface/MAC)

- GbE (Interface/MAC)
- Interlaken (Interface/MAC)
- mPIPE SERDES Control

TRIO / PCIe

- TRIO
- PCIe Interface (SERDES control, endpoint vs. root etc.)
- PCIe Endpoint
- PCIe Root Complex
- PCIe SERDES Control

USB

- USB Host
- USB Endpoint
- USB Host MAC
- USB Endpoint MAC

CHAPTER 4 DDR3 MEMORY CONTROLLERS

The TILE-Gx™ processor has two identical independent memory channels with on-chip memory controllers. Each controller supports the following features:

- Up to 800 MHz memory clock and 1600 MT/s data rate
- 64 bits of data plus optional 8 bits of ECC
- Supports x4, x8, and x16 devices
- Supports up to 16 ranks
- ECC supports (single bit correction, double bit detection)
- Fully programmable DRAM parameters

4.1 Memory Striping

The memory controller striping domain is introduced so that memory workloads are balanced between the memory controllers within the same striping domain. Memory striping can be enabled or disabled.

The load balancing decision is determined by a hash function, based on the values of various address bits. The hash function is configurable to control striping granularity, for example, a stripe can be placed every 512 bytes or every 8192 bytes, and so on.

4.2 Rank/Bank Hashing

Memory requests from different tiles can access different pages of the same DRAM bank for a short period of time. To spread requests across rank/banks, memory controller applies one hashing function. The hashing function is configurable and can be disabled.

4.3 Memory Request Scheduling

To optimize for memory bandwidth and latency, the memory controller uses a 32-entry CAM. The controller looks at all memory requests in the CAM for scheduling. The controller is located between the tiles and the external DRAMs. The controller reorders memory requests, if necessary. On one side, the controller tries to avoid memory request starvation from many tiles. On the other side, the controller tries to reduce the DRAM access overhead (for example from precharge, activation, turnaround, and so on).

4.4 Page Management Policy

DRAM page management policy is a selectable, closed page policy or open page policy. If the closed page option is selected, the controller uses the DRAM auto-precharge feature so that the DRAM page will close after each read or write access. The closed page policy consumes more DRAM power as the result of the activation and precharge commands.

If the open page is selected, the controller, in general, leaves the DRAM page open after each read or write memory access, especially when memory requests with spatial and temporal locality are made. The open page policy is adaptive; it optimizes for memory requests with random addressing. When the memory controller detects DRAM page conflicts, it closes the page ahead of time (for example a precharge command from an available command cycle, or an auto-precharge command is invoked).

4.5 Priority Control

With many tiles, many memory requests can be in-flight at times. Some memory requests are latency sensitive, while some are less critical. A high priority can be assigned to latency-sensitive requests.

The memory controller implements a priority control list in order to control latency conflicts. The priority control list is configurable by system software. Each memory controller has its own priority control list.

Memory controller filters the priority control list and determines the priority level for each memory requests at the run time, for example, read requests from this tile, write requests from that I/O device, instruction reads, and so on.

4.6 Starvation Control

While priority control reduces memory latency on mission critical requests, starvation avoidance helps to constrain an upper bound on latency insensitive requests. If a memory request is pending in a scheduler queue for a long time, then this queue is considered to be “starved”, and a higher priority will be assigned to this queue.

4.7 Performance Counters

To support performance tuning, performance counters are provided for statistics of various events, for example, memory request latency, from request to response within the memory controller, can be measured by a performance counter.

CHAPTER 5 HARDWARE ACCELERATORS

5.1 Overview

This section provides an overview of the TILE-Gx™ Multicore iMesh Coprocessing Accelerator (MiCA™).

MiCA provides a common front-end application interface to off-load or acceleration functions, and performs operations on data in memory. The exact set of operations that it performs is dependent on the specific MiCA implementation. For instance, the TILE-Gx contains two implementations of MiCA architecture, one for cryptographic operations and the other for compression/decompression operations. The architecture is extensible for other types of functions in future generation products.

The MiCA uses a memory-mapped I/O (MMIO) interface to the array of tiles. Because it uses the MMIO interface, access to the MiCA control registers can be controlled through the use of in-tile TLBs. The memory mapped interface enables tiles to instruct the MiCA to perform operations from user processes in a protected manner. Memory accesses performed by the MiCA are validated and translated by an I/O TLB, which is located in the MiCA. This allows completely protected access for operations that user code instructs the MiCA to execute. Since the MiCA system supports virtualized access, each tile is afforded a private view into the accelerator and dozens of operations may be in flight at a given time.

The MiCA connects to TILE-Gx's memory networks and processes requests, which come in via its memory mapped I/O interface. A request consists of pointers to data in memory (a Source Data Descriptor, a Destination Data Descriptor, and an optional Pointer to Extra Data (ED)), a Source Data Length, and an Operation to perform. Many requests can be in flight at one time as the MiCA supports a large number of independent Contexts, each containing their own state. An operation is initiated by writing the request parameters to a Context's User registers.

As the operation progresses, the MiCA verifies that the memory that is accessed by the operation can be accessed legally. If the operation instructs the MiCA to access data that is not currently mapped by the Context's I/O TLB, a TLB Miss interrupt is sent to the Context's bound tile. It is the responsibility of the tile TLB miss handler to fill the I/O TLB. It is also possible for the tile to pre-load the I/O TLB before initiating the request, such that no TLB Misses will occur. At the completion of the operation the MiCA sends a completion interrupt to the Context's bound tile.

Because the MiCA is multi-contexted, multiple operations can be serviced at the same time. Each MiCA implementation has processing engines (for example, crypto, compression, etc.) and a scheduler, which assigns requesting Context's to those engines. All Contexts are independent from each other. Under typical operation, a Context is allocated to a particular tile and that tile then instructs operations of the Context.

A Context is not multi-threaded. If a tile needs overlapped access to a MiCA accessible accelerator, multiple Contexts can be utilized by a single tile.

The MiCA Block Diagram (Figure 5-1) shows a high level view of the MiCA architecture. Descriptions of each of the sub-blocks are provided in the following section.

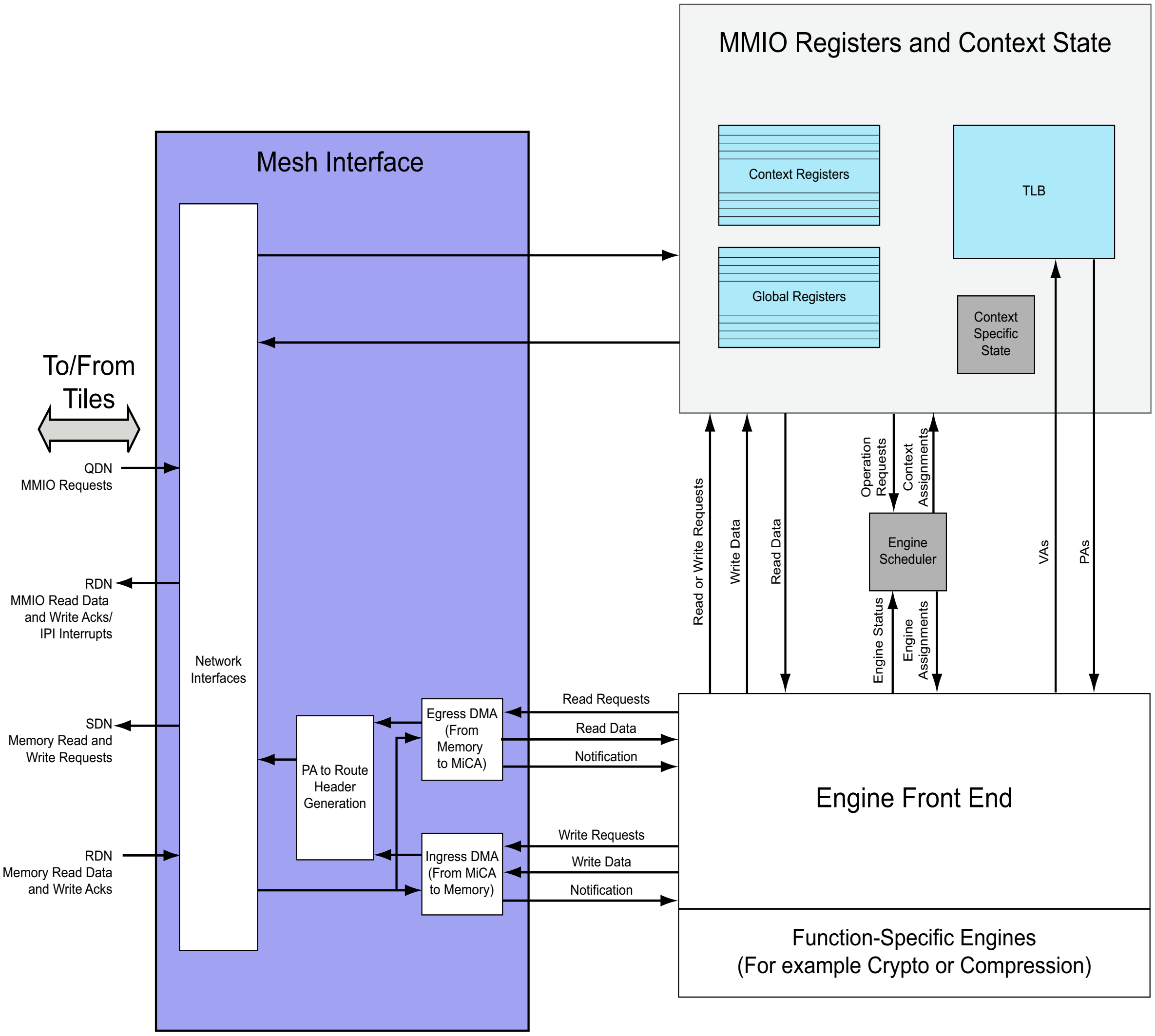
MMIO Registers and Context State
Context Registers
Global Registers
TLB
Context Specific State
Mesh Interface
Network Interfaces
To/From Tiles
QDN MMIO Requests
RDN MMIO Read Data and Write Acks/ IPI Interrupts
SDN Memory Read and Write Requests
RDN Memory Read Data and Write Acks
PA to Route Header Generation
Egress DMA (From Memory to MiCA)
Ingress DMA (From MiCA to Memory)
Read Requests
Read Data
Notification
Write Requests
Write Data
Notification
Read or Write Requests
Write Data
Read Data
Operation Requests
Context Assignments
Engine Scheduler
Engine Status
Engine Assignments
VAs
PAs
Engine Front End
Function-Specific Engines (For example Crypto or Compression)

Figure 5-1. MiCA Block Diagram

5.1.1 Mesh Interface

The MiCA interfaces to the Tiles via the mesh interface. These interfaces allow tile software to access MiCA internal registers, and MiCA to access memory.

Tile access and control of the operation of the MiCA is provided via a set of memory mapped registers. Tiles access the registers via MMIO writes and reads to setup operations and check status.

5.1.2 TLB (Translation Lookaside Buffer)

The TLB is used to store VA-to-PA (Virtual Address-to-Physical Address) translations. It is partitioned per Context. Tiles write to and read from the TLB and can initiate probes to it. The MiCA performs lookups in TLB to translate source, destination, and extra data.

5.1.3 Engine Scheduler

Scheduling consists of assigning Contexts that have operations to perform to hardware resources to perform them (for example, the Function-Specific Engines that do encryption or compression). This function is necessary because there are many more Contexts than Engines. Once an Engine is scheduled to a Context, it completes the operation, that is, without being time-shared within an operation.

5.1.4 Function-Specific Engines

Each MiCA contains multiple Processing Engines, each one is capable of performing a given operation — for example, for compression, decompression, etc. The exact list and number of instances of each type is specific to each MiCA implementation, and typically is much lower than the number of Contexts.

5.1.5 DMA Channels

The DMA channels move data between memory and the Function Specific Engines. Egress DMA is for reading data from memory (packet data and related operating parameters), and Ingress DMA is for writing packet data to memory (note that this convention is the same as for the mPIPE, where Ingress packets travel from external interface into memory, and Egress packets travel from memory to the external interface).

Each Engine has dedicated DMA Egress and Ingress channels assigned to it, so that no Engine is blocked by any other.

5.1.6 PA-to-Header Generation

This block takes the physical address and page attributes from a DMA channel and converts them into a route header to pass to the mesh interface.

5.1.7 Operation

Each MiCA operation reads Source Data from memory, operates on it, and then writes out results to memory as Destination Data. Source data can be specified as either a section of virtual memory, or as a chained buffer as created by mPIPE. Destination data can be specified as either a section of virtual memory, a mPIPE Buffer Descriptor, or a list of mPIPE Buffer Descriptors. The operations can also optionally read and/or write Extra Data that will be needed by an operation. Some operations done in MiCA might not need any Extra Data. For example, encryption keys are specified in Extra Data; a memory-to-memory copy does not use any Extra Data. When used, the Extra Data is specified by its Virtual Address.

The flow of an operation through MiCA is:

1. Tile software or hardware puts source data in memory. For example, the data could be a packet received by the mPIPE.
2. Tile software allocates memory for destination data.
3. Tile software puts extra data, if needed, in memory.
4. Tile software writes parameters describing the operation into its allocated Context Registers.
5. The Context requests use of an Engine from the Scheduler.
6. When an Engine is available, the Scheduler assigns a waiting Context to it.
7. The Engine front end reads operation parameters from the Context's registers.
8. The Engine front end accesses data from memory and sends it to the Engine.

9. Source data is processed by the Engine (for example, encrypted/decrypted, compressed/decompressed, etc), and output is passed to Engine front end and then written to Destination.

   Note: Engine front end performs TLB lookups to translate VAs, as needed. TLB misses generates an IPI interrupt to the Tile (if not masked).

10. When all Destination data has been written to memory, MiCA interrupts the tile. The interrupt also acts as a memory fence; it is not sent until the destination data is visible in memory.

5.1.8 Crypto Accelerators

The MiCA crypto accelerator supports a rich suite of cryptographic algorithms to enable many security protocols, including:

- MACsec (802.1AE)
- IPsec
- SRTP
- SSL/TLS
- Wireless GSM and 3GPP

The MiCA engine supports a range of modes from simple encryption/decryption to complex “packet processing” sequences that offload many of the security protocol steps. However, bulk encryption or proprietary security protocols are also easily supported since the encapsulation and header/trailer processing can be completely handled in tile software.

For key generation, a true entropy-based Random Number Generator (RNG) is incorporated on-chip and it includes a FIPS-140-2 certified whitening function.

The TILE-Gx8036 MiCA system delivers up to 30Gbps of AES encryption performance (128-bit key, CBC or GCM mode).

The symmetric encryption algorithms supported are:

Table 13. Supported Symmetric Encryption Algorithms

Encryption Type	Key Size	Modes
AES	128, 192, 256-bits	CBC, GCM, CTR, ECB
3DES	56, 168-bits	CBC, ECB
ARC4	40 - 256-bits	Stream cipher
KASUMI	128-bits	Stream cipher
SNOW 3G	128-bits	Stream cipher

The cryptographic hashing algorithms supported are:

Table 14. Supported Hashing Encryption Algorithms

Algorithm	Modes
MD5	Straight hash, HMAC
SHA-1	Straight hash, HMAC
SHA-256 (SHA-2)	Straight hash, HMAX

The public key algorithms supported include:

Table 15. Supported Public key Algorithms

Algorithm	Key Sizes
RSA (w/ without CRT)	Up to 4096-bit
DSA	Up to 2048-bit
Elliptic Curve	Up to 1024-bit
ECDSA	Up to 1024-bit
Diffie-Hellman	Up to 2048-bit

5.1.9 Compression Accelerators

Lossless data compression is accelerated through a high-performance “Deflate” compress/decompress engine. Like the MiCA engine, the compression system DMAs data from and to coherent tile memory with no Tile Processor overhead.

The compression acceleration capability is intended to assist with protocols such as:

- IPcomp
- SSL/TLS compression
- gzip

The implementation is fully GZIP compliant with support for dynamic Huffman tables and a 32KB history depth.

The TILE-Gx36 compression system delivers up to 10Gbps of deflate compression performance plus up to 10Gbps of decompression. As there are separate compress and decompress engines, both compress and decompress operations may run simultaneously.

5.1.10 MemCopy DMA Engine

Each of the MiCA engines also incorporates a high-performance DMA capability that can be useful to accelerate memory copy operations for example. It is useful to think of this as a “Null Crypto” or “Null Compress” operation, as the API is virtually identical.

CHAPTER 6 PCIE/TRIO

6.1 PCIe Interfaces

The TILE-Gx8036 incorporates three Gen2 (PCIe 2.0) PCI Express (PCIe) interfaces on-chip. Each of these can be operated in either “Root Complex” or “Endpoint” modes, allowing the TILE-Gx36™ either to be the PCIe master or to appear as an add-in device to a host system.

One of the PCIe interfaces provides 8-lanes and negotiates back to x4, x2 or x1. The raw throughput of this interface in x8 mode is 32 Gbps after the 8b/10b encoding.

The other two PCIe interfaces are configured as 4-lanes and can negotiate back to x2 or x1. They each deliver up to 16 Gbps of raw throughput.

Note that a total of 12-lanes of PCIe SerDes are provided, allowing three ports configured x4 or one port x8 plus one port x4.

CHAPTER 7 XAUI/MPIPE

7.1 mPIPE Subsystem

Packet streams from any of the Ethernet ports or optionally from the PCIe ports are fed into the mPIPE™ subsystem for Packet Classification, Load Balancing, and Buffer Management and Distribution. This block is designed to parse packet headers, perform programmable de-capsulation, filtering, and hashing on selected fields for the purpose of load-balancing flows across "worker tiles". A flexible hardware packet distribution engine works together with a hardware buffer manager to deliver packet streams coherently into the virtual memory system of the chip for access by the designated Tile(s). Packets can be delivered directly to Tile cache or into DDR3 memory as appropriate.

The packet classification engine is C-programmable and can be tailored for a wide ranging field of protocol types and filtering/hashing rules. The packet processing subsystem is designed to handle "wire-speed" packet traffic from the smallest packets (64-bytes) to large Jumbo frames (10Kbytes) at 40Gbps full duplex. For small packets, this means 60 million packets-per-second ingress and 60 million packets-per-second egress.

The packet I/O system supports IEEE 1588 v2 time synchronization protocol. In addition, packets are time-stamped upon arrival with a 64-bit nanosecond-level timer.

IEEE 802.1Qbb priority flow-control is supported.

CHAPTER 8 OTHER I/OS (USB, ETC.)

8.1 USB Subsystem

The TILE-Gx™ Universal Serial Bus (USB) 2.0-compliant system includes two host controllers and one device endpoint controller. Two sets of the UTMI+ Low Pin Interface (ULPI) interface connect the system to the external PHYs. One set of the ULPI interfaces is either used by the host controller or the device endpoint controller, and the other ULPI interface is dedicated to the second host controller.

One host controller system consists of one Enhanced Host Controller Interface (EHCI) core and one Open Host Controller Interface (OHCI) core. It accesses the memory system via the iMesh™ network with 64-bit addressing capability. The addresses are translated to physical addresses using a Translation Lookaside Buffer (TLB) that supports all the standard TILE-Gx I/O-TLB attributes.

The device endpoint controller supports one configuration and up to four interfaces. In addition to the default endpoint 0, seven extra sets of endpoint functions are provided. Data transfers between the main memory and the endpoints are controlled by the Tile processors using the memory-mapped I/O (MMIO) interface. The device endpoint controller can also be used in the boot and debug the TILE-Gx under the special mode operation.

8.2 Flexible I/O System

The TILE-Gx flexible I/O system consists of 64 data pins that are capable of configuring and formatting data to implement low-speed status and control bits, or implementing moderate speed asynchronous interface protocols such as HPI, ATA, etc. Each I/O pin can be individually configured to be an input, output, or bidirectional pin with a number of drive and input options.

The system supports simultaneous use by multiple processes with full protection and virtualization support. The virtualization support allows direct access to the interface by application-level programs with full process isolation and protection. MMIO transfers are used to configure the interface and to supply and receive data from the I/O pins. An interrupt capability is supplied to allow interrupts to be generated on any transition of a pin.

8.3 UART System

The TILE-Gx UART system communicates the processors with the external device via the two UART serial bits. The system can operate in two modes: interrupt mode and protocol mode.

In the interrupt mode, the UART controller provides a typical transmit/receive interface between an external device and on-chip processors. Data written to the write FIFO by a processor is transferred to an internal transmit FIFO and then transmitted out the serial transmit output. Data received on the serial receive input is transferred to the receive FIFO, which the processor can then read.

In the protocol mode, the UART controller provides an external devices with the ability to read or write any register in any TILE-Gx RSHIM devices. Bytes received via the serial receive input are interpreted as register reads or write commands. Read responses are transmitted via the serial transmit output.

8.4 I^{2}C Systems

The I^{2}C Master Interface provides an interface for processors to write and read an external I^{2}C devices. It is capable of reading from an external EEPROM at boot and writing to any Rshim device registers.

The I^{2}C Slave interface is the interface to an external initiator (master). The interface supports Standard-mode, Fast-mode, and Fast-mode plus. The I^{2}C slave controller also supports clock stretching.

8.5 SPI System

The SPI SROM system provides an interface for processors to write and read an off-chip SPI SROM. It also includes a hardware state machine that can read from the SPI SROM at boot time and then write any Rshim device registers.

CHAPTER 9 DEBUG

Debugging of applications software typically uses industry-standard methods, such as the GNU debugger (GDB).

In addition each Tile provides hardware diagnostics, and performance monitoring capabilities:

- Tile Timer

  A 31-bit down counter with an interrupt is provided in the Tile. The interrupt interval is programmable and can be used for operating system level "tick" functionality or for any other timing task.

- Cycle Counter

  The Tile provides a 64-bit free running cycle counter; the counter initializes to 0 but can be changed by privileged software. It can also be read by software at any privilege level.

- Events

  The performance monitoring and system debug capabilities of the Tile architecture rely on implementation-defined events. The specific set of events available to software varies depending on implementation, but examples include cache-miss, instruction bundle retired, network word sent and so on. Events are used to increment performance counters or interact with system debug/diagnostics functionality.

- Counters

  The Tile architecture provides four 32-bit performance counters. The counters can be assigned to any one of the implementation specific events. On overflow, the counter triggers an interrupt.

- Watch Registers

  The Tile architecture provides programmable watch registers to track matches to implementation specific multi-bit fields. For example, match on a specific fetch PC, or a specific memory reference Virtual Address.

- Tile Debug Port

  This feature aids with system software debugging, TILE-Gx™ provides access to essential processor state data such as fetch-PC, registers, SPRs, and caches.

CHAPTER 10 BOOT

Booting the TILE-Gx™ device is fundamentally a three-step process.

The level-0 boot code is built into a ROM in each Tile, and consists of a small program that receives the level-1 boot code from an external device such as a serial ROM, I^2C master (the TILE-Gx™ is the slave), flash memory, USB, or PCIe host.

The level-1 boot performs primary device initialization functions including memory controller configuration, physical address space mapping, and local I/O device discovery.

Once level-1 boot is complete, the level-2 boot (remaining Hypervisor, OS, application) can be performed over any of TILE-Gx interfaces including PCIe, Ethernet, UART, I^2C, or Flexible I/O. The level-1 boot code must initialize the level-2 boot interface to enable the level-2 boot process.

G GLOSSARY

Term	Definition
BARs	Base address registers.
BIST	Built in Self Test.
CAM	Content Addressable Memory.
CPLD	Complex PLD. A programmable logic device (PLD) that is made up of several simple PLDs (SPLDs) with a programmable switching matrix in between the logic blocks. CPLDs typically use EEPROM, flash memory or SRAM to hold the logic design interconnections.
DDC™	Dynamic Distributed Cache. A system for accelerating multicore coherent cache subsystem performance. Based on the concept of a distributed L3 cache, a portion of which exists on each tile and is accessible to other tiles through the iMesh. A TLB directory structure exists on each tile - eliminating bottlenecks of centralized coherency management - mapping the locations of pages among the other tiles.
ECC	Error-Correcting Code. A type of memory that corrects errors on the fly.
fabric chaining	The ability to cascade multiple Tilera chips together seamlessly in order to provide more processing power, memory, and I/O for an application. The architecture is designed to allow fabric chaining to be done transparently to the application such that major software rewrites are unnecessary.
hardwall technology	A microcode feature that can partition a Tile Processor into multiple virtual machines, allowing different instances of Linux and their applications to run on the chip and be isolated from each other.
host port interfaces (HPIs)	A 16-bit-wide parallel port through which a host processor can directly access the CPU’s memory space. The host device functions as a master to the interface, which increases ease of access. The host and CPU can exchange information via internal or external memory. The host also has direct access to memory-mapped peripherals. Connectivity to the CPU's memory space is provided through the DMA controller.
Hypervisor services	Provided to support two basic operations: install a new page table (performed on context switch), and flush the TLB (performed after invalidating or changing a page table entry). On a page fault, the client receives an interrupt, and is responsible for taking appropriate action (such as making the necessary data available via appropriate changes to the page table, or terminating a user program which has used an invalid address).

Term	Definition
Interpacket Gap (IPG)	Ethernet devices must allow a minimum idle period between transmission of Ethernet frames known as the interframe gap (IFG) or interpacket gap (IPG). It provides a brief recovery time between frames to allow devices to prepare for reception of the next frame.
MDIO	Management interface I/O bidirectional pin. The management interface controls the behavior of the PHY.
MiCA™	Multistream iMesh Crypto Accelerator engines. The MiCA engines include a robust set of encryption, hashing, and public key operations.
MMIO	Memory-Mapped I/O.
Multicore Development Environment™ (MDE)	Multicore software programming environment.
promiscuous mode	In computing, refers to a configuration of a network card wherein a setting is enabled so that the card passes all traffic it receives to the CPU rather than just packets addressed to it, a feature normally used for packet sniffing. Many operating systems require superuser privileges to enable promiscuous mode. A non-routing node in promiscuous mode can generally only monitor traffic to and from other nodes within the same collision domain (for Ethernet and Wireless LAN) or ring (for Token ring or FDDI).
RGMII	Reduced Gigabit Media Independent Interface.
SPI ROM	Serial Flash with serial peripheral interface.
TRIO interface	Transaction IO. The TRIO interface provides DMA and other data movement services for “read/write” protocols such as PCIe, SRIO, and streaming IO.
UART	(Universal Asynchronous Receiver Transmitter). The electronic circuit that makes up the serial port. Also known as “universal serial asynchronous receiver transmitter” (USART), it converts parallel bytes from the CPU into serial bits for transmission, and vice versa. It generates and strips the start and stop bits appended to each character.
VLIW architecture	VLIW (Very Long Instruction Word). A microprocessor design technology. A chip with VLIW technology is capable of executing many operations within one clock cycle. Essentially, a compiler reduces program instructions into basic operations that the processor can perform simultaneously. The operations are put into a very long instruction word that the processor then takes apart and passes the operations off to the appropriate devices.

I INDEX

A
AES encryption performance **34**
algorithms
  cryptographic hashing **35**
  public key **35**
  symmetric encryption **34**
ALU
  destination **5**
  operations **6**
ATA **41**
atomic
  memory operations **10**
  operations **9**

B
Ball Grid Array
  See BGA
BAR **47**
base address registers
  See BAR
BGA **1**
BIST **47**
bit field insert/extract **4**
block diagram
  cache subsystem **10**
branch predict **5**
  stage **6**
bulk encryption **34**
byte shuffle **4**

C
C intrinsics **4**
cache
  operation **12**
cache-coherent approach to shared memory **12**
CAM **47**
clock rate **1**
closed page policy **29**
cmpexch **10**
CmpXchg **9**
common
  registers **25**
complex
  multiply **3**
  PLD **47**
compression accelerators **35**
conditional
  branches **6**
  result support **4**
context's user registers **31**
CPL **15**
CPLD **47**
crypto accelerators **34**
cryptographic hashing algorithms, listed **35**

D
DDC **12**, **47**
DDR3 memory
  packet processing **39**
decode **5**
  pipeline stage **5**
  stage **6**
detail of a Tile within the Tile Processor **3**
device specific registers **26**
DMA channels **33**
DRAM auto-precharge feature **29**
DSP
  and SIMD instructions **3**
  applications **11**
DTLB **13**
  hazard **5**
Dynamic Distributed Cache **47**
Dynamic Distributed Caching
  See DDC

E
ED **31**
encapsulation **34**
Endpoint mode **37**
engine scheduler **33**
engines
  function-specific **33**
Enhanced Host Controller Interface (EHCI) **41**
exceptions **13**
exch **10**
execute0 **6**
execute1 **6**
  pipeline stage **6**
execution units/pipelines **8**
extra data
  See ED

F
fabric chaining **47**
FABRIC_CONN register **25**
fetch **4**
fetchadd **9**, **10**
fetchaddgez **9**, **10**
fetchand **9**, **10**
fetchor **9**, **10**
flexible I/O system **41**
flits **18**
front end **7**
function-specific engines **33**

G
GCM mode **34**
gzip **35**

H
hardwall technology
  defined **47**
hardware buffer manager **39**
hashing function **29**
header **18**
home tile **12**
host port interfaces
  See HPIs
how to set protection levels **16**
HPI **41**, **47**
  defined **47**
Huffman tables **35**
Hypervisor services **47**

I
I/O connectivity **25**
I2C slave interface **42**
icoh **5**
  instruction **9**
IFG **48**
iMesh **16**
  interconnect **16**
independent contexts **31**
instruction
  execution
    preventing illegal instructions **14**
  memory **9**
  set
    overview **3**
instruction set architecture
  See ISA
instructions
  DSP and SIMD **3**
interframe gap
  See IFG
interpacket gap
  See IPG
Interprocess Interrupt
  See IPI
interrupt
  number
    See also IPI MPL **24**
interrupt mode **41**
interrupts **13**
INTERRUPT_VECTOR_BASE SPR **14**
IPcomp **35**
IPG **48**
IPI **24**
  information **25**
IPI MPL **24**
IPsec **34**
iret **14**
ISA **3**, **14**
I-stream prefetch operation **8**
ITLB **13**

K
key generation **34**

L
L2 cache subsystem **3**
laned
  compare **3**
  multiply **3**
large Jumbo frames **39**
load
  balancing **39**
  destination **5**

M
MACsec **34**
MDE **48**
MDIO **48**
MemCopy DMA engine **35**
memory
  controller striping domain **29**
  fence **5**
  fence instruction **9**
    See mf
  management units (MMUs) **24**
  mapped I/O
    See MMIO
  ordering **9**
  request latency **30**
  system **1**
MEMORY_ATTRIBUTE **23**
mesh interface **32**
mf **10**
mfspr **14**
MiCA **31**, **48**
  block diagram **32**
  operation flow through **33**

minimum protection level
    See MPL
MMIO **9**, **23**, **31**, **41**, **48**
  transfers **41**
MMUs **24**
mPIPE subsystem **39**
MPL **13**, **15**
MPL_BOOT_ACCESS **16**
MPL_ITLB_MISS **16**
mtspr **14**
Multicore iMesh Coprocessing Accelerator
    See MiCA

N
nap instruction **4**, **5**
network mapped GPR **5**
Network Priority Arbitration (NPA) **19**
Null Compress operation **35**
Null Crypto operation **35**

O
on-chip mesh interconnects **3**
Open Host Controller Interface (OHCI) **41**
open page **30**
  policy **29**
operation flow through MiCA **33**
operations
  Null Compress **35**
  Null Crypto **35**

P
packet
  classification **39**
  headers
    parsing **39**
  processing subsystem **39**
page tables **9**
PA-to-header generation **33**
PC **4**, **14**
PCB designs **1**
PCIe
  master **37**
  ports **39**
performance
  counters **30**
  tuning **30**
physical addresses **9**
pipeline latencies **6**
population count **4**
preventing illegal instruction execution **14**
priority control list **30**
privileged SPR **8**
processing engines **31**
processor core **3**

program counter
    See PC
promiscuous mode
  defined **48**
protected_resource **16**
protection
  architecture **14**
  levels **15**
    how to set **16**
  mechanisms **14**
protocol mode **41**
public key algorithms, listed **35**

R
Random Number Generator
    See RNG
RAS **5**, **8**
register file **6**
registers **3**
return address stack
    See RAS
RGMII **48**
RNG **34**
RoHS-6 compliant **1**
root complex mode **37**
round robin arbitration (RRA) **19**
RSH_COORD SPR **25**
RSH_DEV_INFO **25**
RSH_FABRIC_CONN register **25**
Rshim device registers **42**
RSHIM_COORD SPR **25**
RSH_IPI_LOC register **25**
run-to-completion process **1**

S
SAD **4**
saturating arithmetic **3**
SBOX_CONFIG SPR **8**
security protocols
  IPsec **34**
  MACsec **34**
  SRTP **34**
  SSL//TLS **34**
  wireless GSM and 3GPP **34**
self-modifying code **9**
SMP **1**
  Linux **1**
software interrupt instruction
    See swint
source operand **6**
Source Operand RAW **5**
Special Purpose Registers
    See SPRs
SPI ROM **48**
SPI SROM **42**

SPRs **4**, **13**
SRTP **34**
SSL/TLS **34**
  compression **35**
standard Linux **1**
sum of absolute differences
  See SAD
swint **13**
switch interface **3**
symmetric encryption algorithms, listed **34**
Symmetric Multi-Processing
  See SMP
synchronization protocol **39**

T
tile
  block diagram showing cache subsystem **10**
  cache **10**
  cache attributes **7**, **10**, **12**
  processor
    cross-section **3**
tile-fabric sizing **25**
TILE-Gx switch interfaces
  illustrated **17**
TILE-Gx8036 **1**
TLB **15**, **23**, **32**, **41**
  lookups **34**
Translation Lookaside Buffer
  See TLB
TRIO interface
  defined **48**

U
UART **41**
  defined **48**
ULPI **41**
Universal Serial Bus
  See USB
USB **41**
UTMI+ Low Pin Interface
  See ULPI

V
VA-to-PA translations **32**
Very Long Instruction Word
  See VLIW
virtual
  addresses **9**
VLIW **3**
  architecture
    defined **48**

W
wireless GSM and 3GPP **34**
write back **6**
  pipeline stage **6**
  stage **6**

X
Xchg **9**