Epiphany Architecture Reference

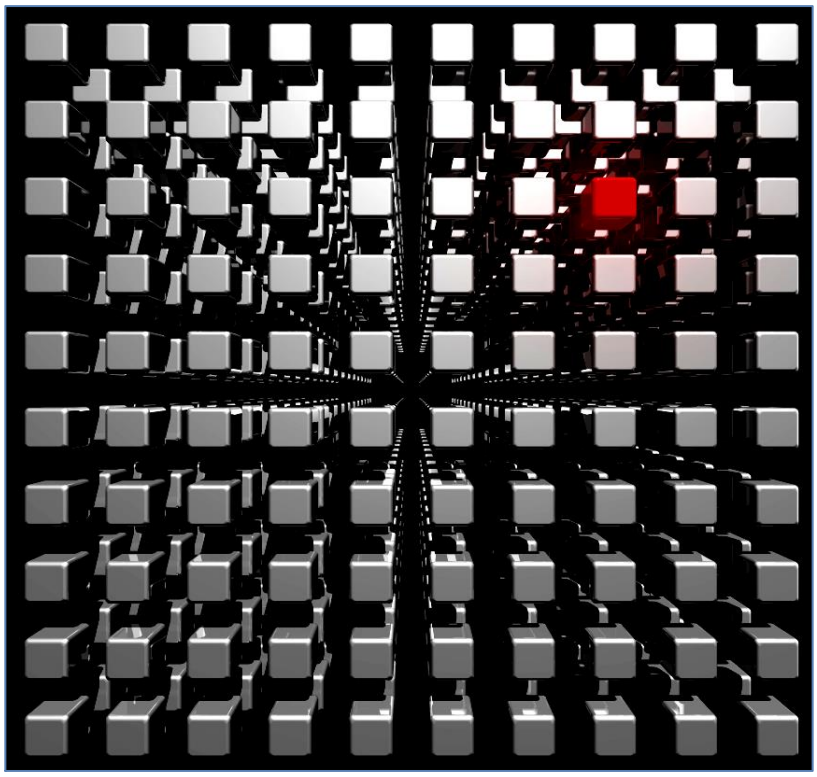


Table of Contents

List of Figures

List of Tables

Preface

This document describes Adapteva's Epiphany™ architecture. The document is written for system programmers with a fundamental understanding of processor architectures and experience with C programming.

Related Documents

- Epiphany SDK Reference: The development tools and run-time library for the Epiphany architecture.
- Epiphany E16G301 Datasheet: Datasheet for 16-core System-on-Chip implementation of the Epiphany architecture.

"LABS" Features

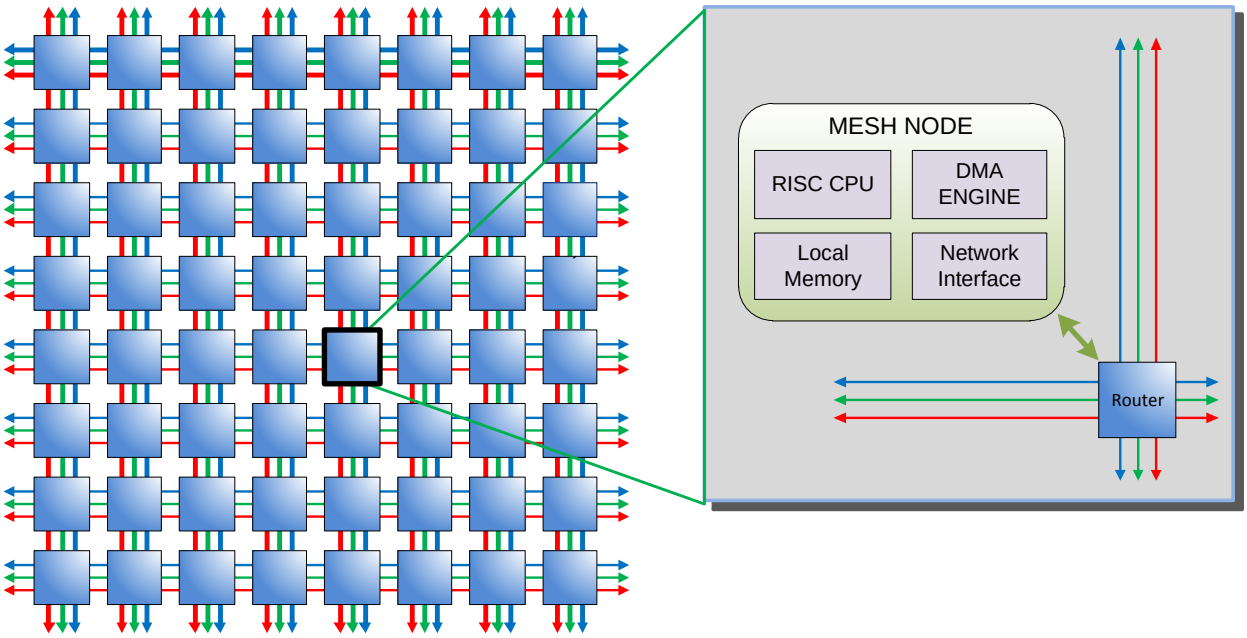
Features labeled with the "LABS" label should be considered experimental.

1 Introduction

The Epiphany architecture defines a multicore, scalable, shared-memory, parallel computing fabric. It consists of a 2D array of compute nodes connected by a low-latency mesh network-on-chip. Figure 1 shows an implementation of the architecture, highlighting the key components:

- A superscalar, floating-point RISC CPU in each mesh node that can execute two floating point operations and a 64-bit memory load operation on every clock cycle.
- Local memory in each mesh node that provides 32 Bytes/cycle of sustained bandwidth and is part of a distributed, shared memory system.
- Multicore communication infrastructure in each node that includes a network interface, a multi-channel DMA engine, multicore address decoder, and network-monitor.
- A 2D mesh network that supports on-chip node-to-node communication latencies in nanoseconds, with zero startup overhead.

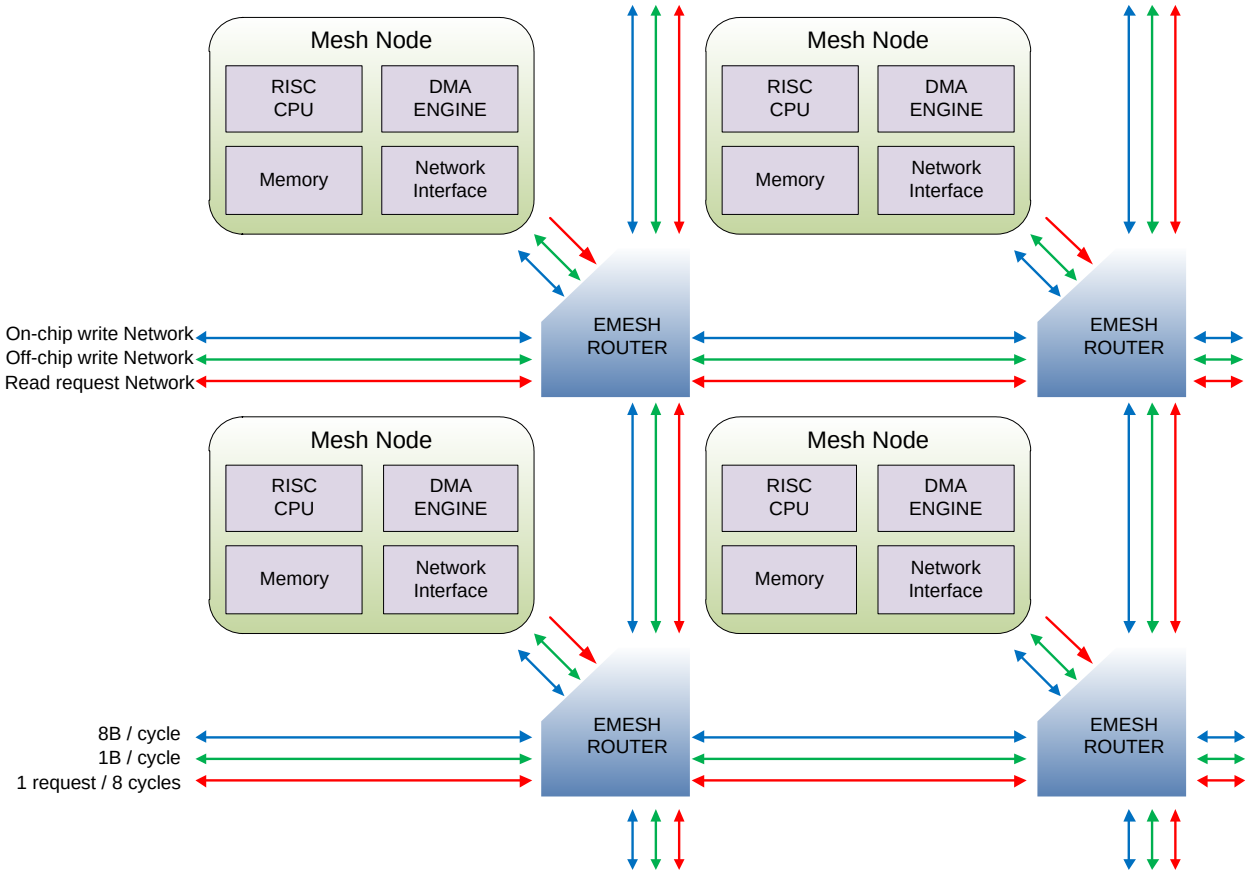
Figure 1: An Implementation of the Epiphany Architecture



The Epiphany architecture was designed for good performance across a broad range of applications, but really excels at applications with high spatial and temporal locality of data and

code. Examples of such application domains include: image processing, communication, sensor signal processing, encryption, and compression. High speed inter-processor communication is supported by the Epiphany architecture's 2D eMesh™ Network-On-Chip (NOC), shown in Figure 2, which connects the on-chip processor nodes. The mesh network efficiently handles traffic patterns in high-throughput real-time applications. The network takes advantage of spatial locality and an abundance of short point-to-point on-chip wires to send complete transactions—consisting of source address, destination address, and data—in a single clock cycle. Each routing link can transfer up to 8 bytes of data on every clock cycle, allowing 64 bytes of data to flow through every routing node on every clock cycle, supporting an effective bandwidth of 64 GB/sec at a mesh operating frequency of 1GHz.

Figure 2: eMesh™ Network-On-Chip Overview



The memory map of the Epiphany architecture is flat and unprotected. Every mesh node has direct access to the complete memory system, without limitation. The architecture employs a flat 32-bit memory map and supports up to 4096 individual mesh nodes.

The shared-memory architecture and low-latency, on-chip mesh network allows multicore programs to pass messages from a few bytes to kilobytes with very little overhead. The high bandwidth and low latency of the eMesh™ NOC means the Epiphany can support parallel programming at a large kernel as well as fine-grained level in which small tasks can be executed in parallel. The support of many different levels of parallelism within the Epiphany architecture is a true breakthrough that will make parallel programming much easier and effective by significantly reducing inter-task communication bottlenecks.

The key benefits of the Epiphany architecture are:

- **Ease of Use:** A multicore architecture that is ANSI-C/C++ programmable. This makes the architecture accessible to every programmer, regardless of his or her level of expertise.

    - **Effectiveness:** The general-purpose instruction, superscalar instruction issue, and large unrestricted register file ensures that the application code written in ANSI-C can approach the peak theoretical performance of the Epiphany architecture.

- **Low Power**: Aggressive microarchitecture optimizations, streamlined feature sets, and extensive clock gating enables up to 70 GFLOP/Watt processing efficiency at 28nm.

- **Scalability:** The architecture can scale to thousands of cores on a single chip and millions of cores within a larger system. This provides the basis for future performance gains from increased parallelism.

2 Programming Model

2.1 Programming Model Introduction

The Epiphany architecture is programming-model neutral and compatible with most popular parallel-programming methods, including Single Instruction Multiple Data (SIMD), Single Program Multiple Data (SPMD), Host-Slave programming, Multiple Instruction Multiple Data (MIMD), static and dynamic dataflow, systolic array, shared-memory multithreading, message-passing, and communicating sequential processes (CSP). Adapteva anticipates that with time, the ecosystem around the Epiphany multicore architecture will grow to include many of these methods.

The key hardware features in the Epiphany architecture that enables effective support for parallel programming methods are:

- General-purpose processors that support ANSI C/C++ task level programming at each node. Shared-memory map that minimizes the overhead of creating task interfaces.
- Distributed-routing technology that decouples tasks from
- Inter-core message-passing with zero startup cost.
- Built-in hardware support for efficient multicore data-sharing.

2.2 Parallel Programming Example

The following example shows how multiple Epiphany mesh nodes can be combined to improve the overall throughput of a computation. For simplicity, we have chosen matrix multiplication, but the concepts also apply to more complicated programs. Matrix multiplication can be represented by the following formula:

$$C_{ij} = \sum_{k=0}^{N-1} \left(A_{ik} \, B_{kj} \right)$$

Where A and B are the input matrices, C is the result, and i and j represent the row-column coordinate of the matrix elements.

A naïve (but correct) implementation of the matrix multiplication running on a single core is given below:

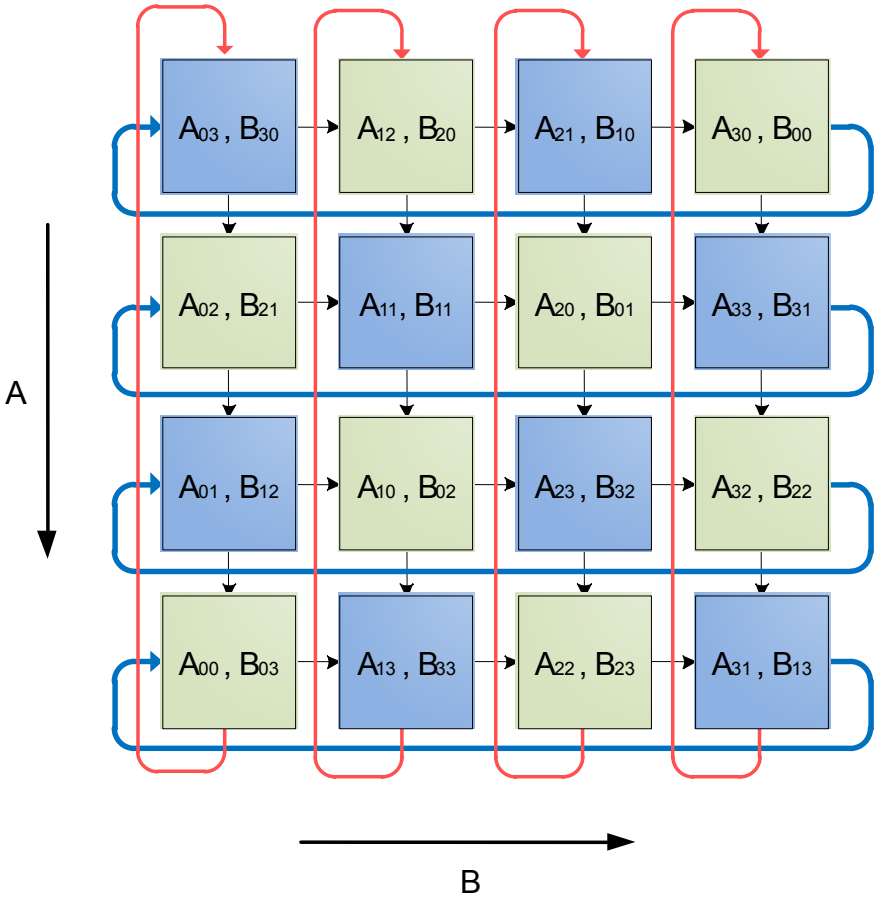
```
for(i = 0; i < M; i++)
     for( j = 0; j < N; j++)
        for( k = 0; k < K; k++)
          C[i][j] += A[i][k] * B[k][j];
```

The code above can be written in standard C/C++ and compiled to run on a single core, with matrices A, B, and C placed in the core's local memory. In this simple programming example, there is no difference between the Epiphany architecture and any other single threaded processor platform.

To speed up this calculation using several mesh nodes simultaneously, we first need to distribute the A, B, C matrices over P tasks. Due to the matrix nature of the architecture, the natural way to distribute large matrices is by cutting them into smaller blocks, sometimes referred to as "blocked by row and column". We then construct a SPMD program that runs on each of the mesh nodes.

Figure 3 shows how the matrix multiplication can be divided into 16 sub-tasks and mapped onto 16 mesh nodes. Data sharing between the sub tasks can be done by passing data between the cores using a message passing API provided in the Epiphany SDK or by explicitly writing to global shared memory.

Figure 3: Matrix Multiplication Data Flow



The parallel matrix multiplication completes in √P steps, (where P is the number of processors) with each matrix multiplication task operating on data sets that are of size √P x √P. At each step of the process, contributions to the local C matrix accumulate in each task, after which the local A matrix moves down and the local B matrix moves to the right. The entire example can be completed using standard ANSI programming constructs. Epiphany run-time functions are provided to simplify multicore programming, but their use is not mandatory. The architecture allows programmers to innovate at all levels. For more information on the inter-processor communication API, please refer to the *Epiphany SDK Reference Manual*.

Given the algorithm above, a 16-core Epiphany implementation operating at 1GHz can complete a 128x128 matrix multiply in 2ms while achieving 90% of the theoretical peak performance. The matrix multiplication algorithm in this example scales to thousands of cores and demonstrates how the Epiphany architecture's performance scales linearly with the number of cores in the system when proper data distribution and programming models are used.

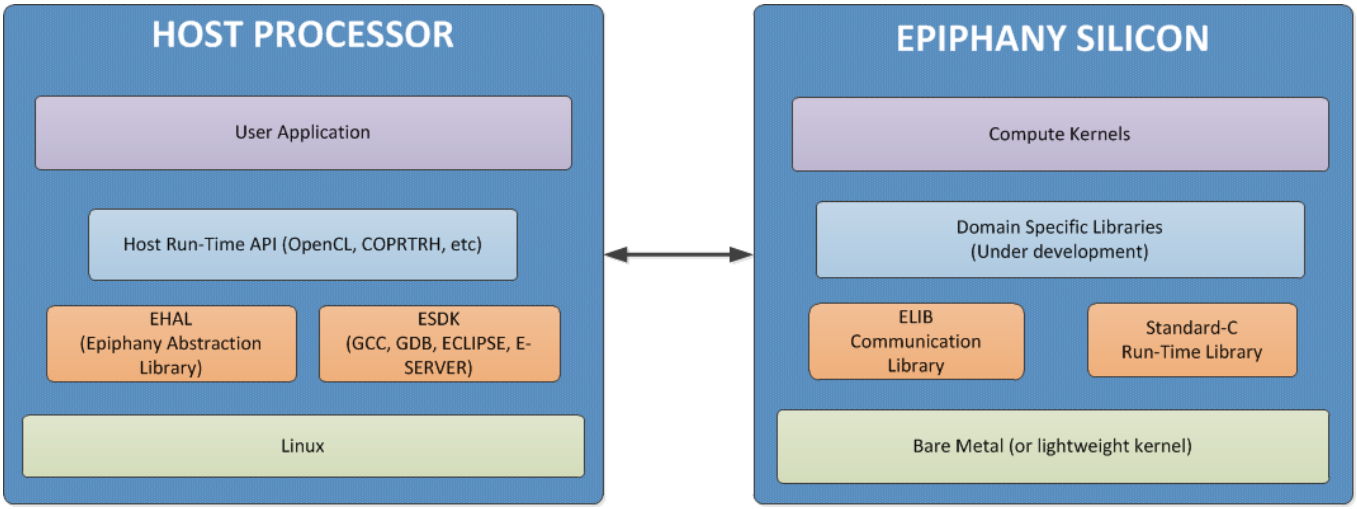
3 Software Development Environment

The Epiphany multicore architecture supports popular open-source ANSI C/C++ software development flows, using GNU GCC and GDB. The highly optimized GCC compiler enables acceptable real-time performance from pure ANSI-C/C++ applications without having to write assembly code for the vast majority of applications. The Epiphany SDK includes:

- ANSI-C/C++ GCC compiler
- OpenCL SDK
- Multicore GDB debugger
- Eclipse based multicore IDE
- Runtime library
- Fast functional single core simulator

Figure 4 shows the complete software stack of the Epiphany software development environment.

Figure 4: Epiphany Software Development Stack

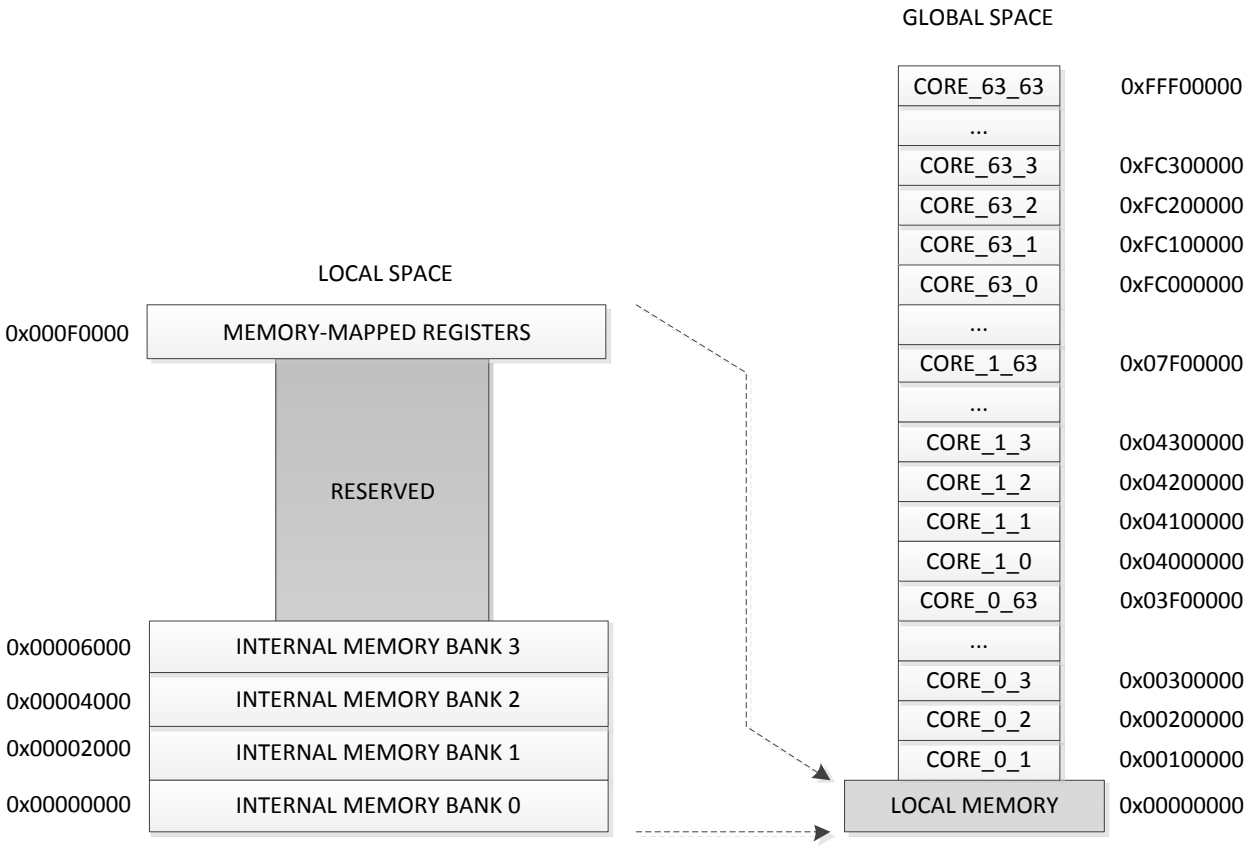


4 Memory Architecture

4.1 Memory Address Map

The Epiphany architecture uses a single, flat address space consisting of 2^{32} 8-bit bytes. Byte addresses are treated as unsigned numbers, running from 0 to $2^{32} - 1$. This address space is regarded as consisting of 2^{30} 32-bit words, each of whose addresses is word-aligned, which means that the address is divisible by 4. The word whose word-aligned address is A consists of the four bytes with addresses A, A+1, A+2 and A+3. Each mesh node has a local, aliased, range of memory that is accessible by the mesh node itself starting at address 0x0 and ending at address 0x00007FFF. Each mesh node also has a globally addressable ID that allows communication with all other mesh nodes in the system. The mesh-node ID consists of 6 row-ID bits and 6 column-ID bits situated at the upper most-significant bits (MSBs) of the address space. The complete memory map for the 32 bit Epiphany architecture is shown in Figure 5.

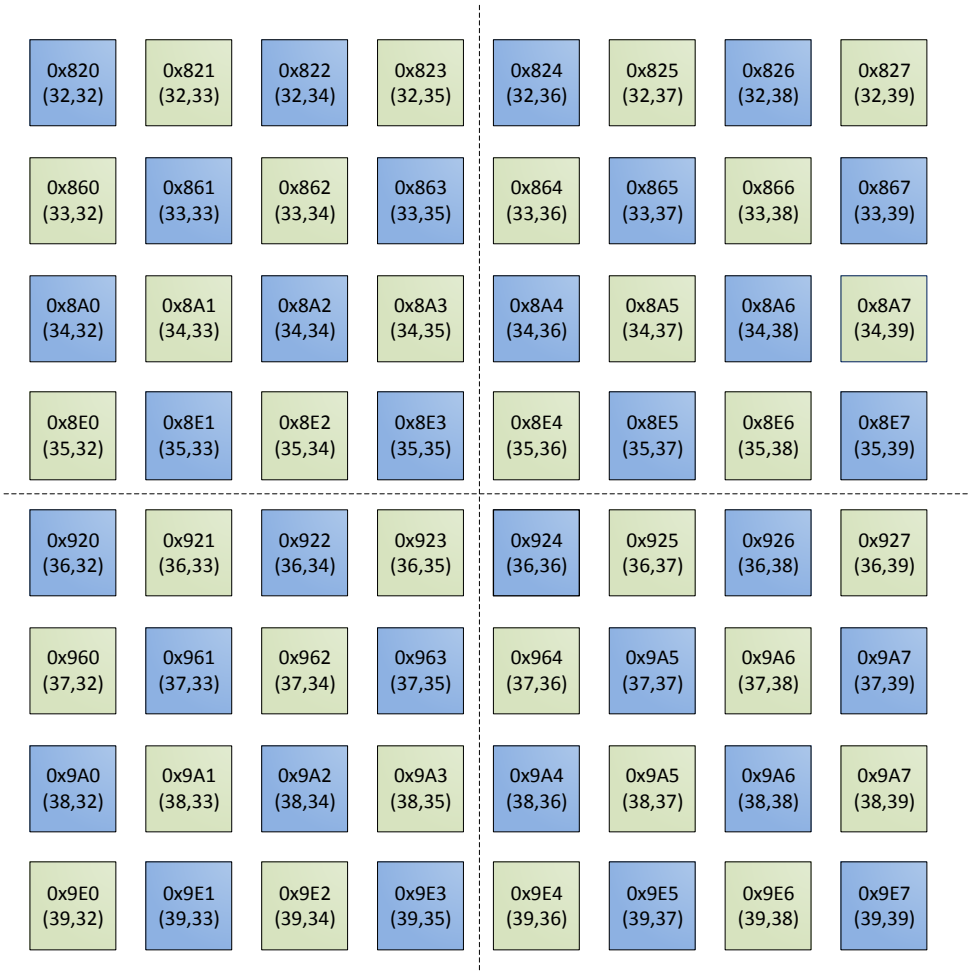
Figure 5: Epiphany Global Address Map



Data and code can be placed anywhere in the memory space or in external space, except for the memory-mapped register space and reserved space, but performance is optimized when the data and code are placed in separate local-memory banks.

Figure 6 shows a 64-node region of the memory map, highlighting the upper address range of each mesh node and its corresponding mnemonic (row, column) coordinate. Note that the memory map laid out as a mesh to match the natural geometrical mapping of Epiphany's Network-On-Chip. The dotted line in Figure 6 shows the I/O boundary and memory map for a hypothetical system consisting of four 16-core chips connected in a glue-less fashion on a board. The 32-bit address map supports up to 4095 cores in a single shared memory system, but practically some of the memory space will probably be dedicated to off-chip SDRAM and memory mapped IO peripherals.

Figure 6: Epiphany Shared Memory Map



Each CPU can be accessed by any other CPU by specifying the appropriate row-column fields of the address in a memory read or write transactions. The startup cost for node-to-node communication is zero clock cycles. From a programmer's viewpoint, the only difference between on-chip communication and off-chip communication is in transaction bandwidth and latency. In the Figure 6 memory map, if core (32,32) wants to perform a read operation from core (39,39), it would send a read address with the upper bits 0x9e7 and specify a return address with upper bits 0x820 to the mesh network. The network takes care of the rest, making sure that the read request propagates to the read destination and that data is correctly returned to the mesh node that initiated the request.

4.2 Memory Order Model

All read and write transactions from local memory follow a *strong memory-order model*. This means that the transactions complete in the same order in which they were dispatched by the program sequencer.

For read and write transactions that access non-local memory, the memory order restrictions are relaxed to improve performance. This is called a *weak memory-order model*. The following section explains the background of a weak memory-order model, how it is used by the Epiphany architecture, and how it affects determinism in the system. The relaxation of synchronization between memory-access instructions and their surrounding instructions is referred to as *weak ordering of loads and stores*. Weak ordering implies that the timing of the actual completion of the memory operations—even the order in which these events occur—may not align with how they appear in the sequence of the program source code. The only guarantees are:

- Load operations complete before the returned data is used by a subsequent instruction.
- Load operations using data previously written use the updated values.
- Store operations eventually propagate to their ultimate destination.

Weak ordering has some side-effects that programmers must be aware of in order to avoid improper system operation. When writing to or reading from non-local memory locations, such as off-chip I/O device registers and SDRAM, the order in which read and write operations complete is often significant, but is not guaranteed by the underlying hardware. To ensure that these effects do not occur in code that requires strong ordering of load and store operations, use run-time synchronization calls with order-dependent memory sequences.

Table **1** shows the ordering guaranteed in the Epiphany architecture. Instruction #1 refers to the first instruction in a sequential program, and instruction #2 refers to any instruction following the first one in that same program.

Table 1: Memory Transaction Ordering Rule

First Transaction	Second Transaction	Deterministic Order
Read from CoreX	Read from CoreX	Yes
Write to CoreX	Write to CoreX	Yes
Write to CoreX	Read from CoreX	No
Read from CoreX	Write to CoreX	Yes
Read from CoreX	Read from CoreY	Yes
Read from CoreX	Write to CoreY	Yes
Write to CoreX	Write to CoreY	No!
Write to CoreX	Read from CoreY	No!

4.3 Endianness

The Epiphany architecture is a little-endian memory architecture. The figures below show how instructions and data are placed in memory with respect to byte order.

Data In Register				Data In Memory			
B3	B2	B1	B0	B3	B2	B1	B0
				Addr+3	Addr+2	Addr+1	Addr+0
32-Bit Instruction In Register				32-Bit Instruction In Memory			
B3	B2	B1	B0	B3	B2	B1	B0
				Addr+3	Addr+2	Addr+1	Addr+0
16-Bit Instruction In Register				16-Bit Instruction In Memory			
B1	B0	B1	B0	B1	B0	B1	B0
				Addr+3	Addr+2	Addr+1	Addr+0

4.4 Load/Store Alignment Restrictions

The Epiphany architecture expects all memory accesses to be suitably aligned: doubleword accesses must be doubleword-aligned, word accesses must be word-aligned, and halfword accesses must be halfword-aligned. Table 2 summarizes the restrictions on the three LSBs of the address used to access memory for each type of memory transaction. An “x” in the address field refers to a bit that can be any value. Load and store transactions with unaligned addresses generate a software exception that is handled by the node's interrupt controller. For unaligned write accesses, data is still written to memory, but the data written will be incorrect. Unaligned reads return values to the register file before an unaligned exception occur.

Table 2: Load and Store Memory-Alignment Restrictions

Data Type	Address[2:0]
Byte	Xxx
Halfword	xx0
Word	x00
Doubleword	000

4.5 Program-Fetch Alignment Restrictions

All instructions must be aligned on half-word boundaries.

5 eMesh Network-On-Chip

The eMesh Network-On-Chip is illustrated in Figure 2 and in Figure 7.

5.1 Network Topology

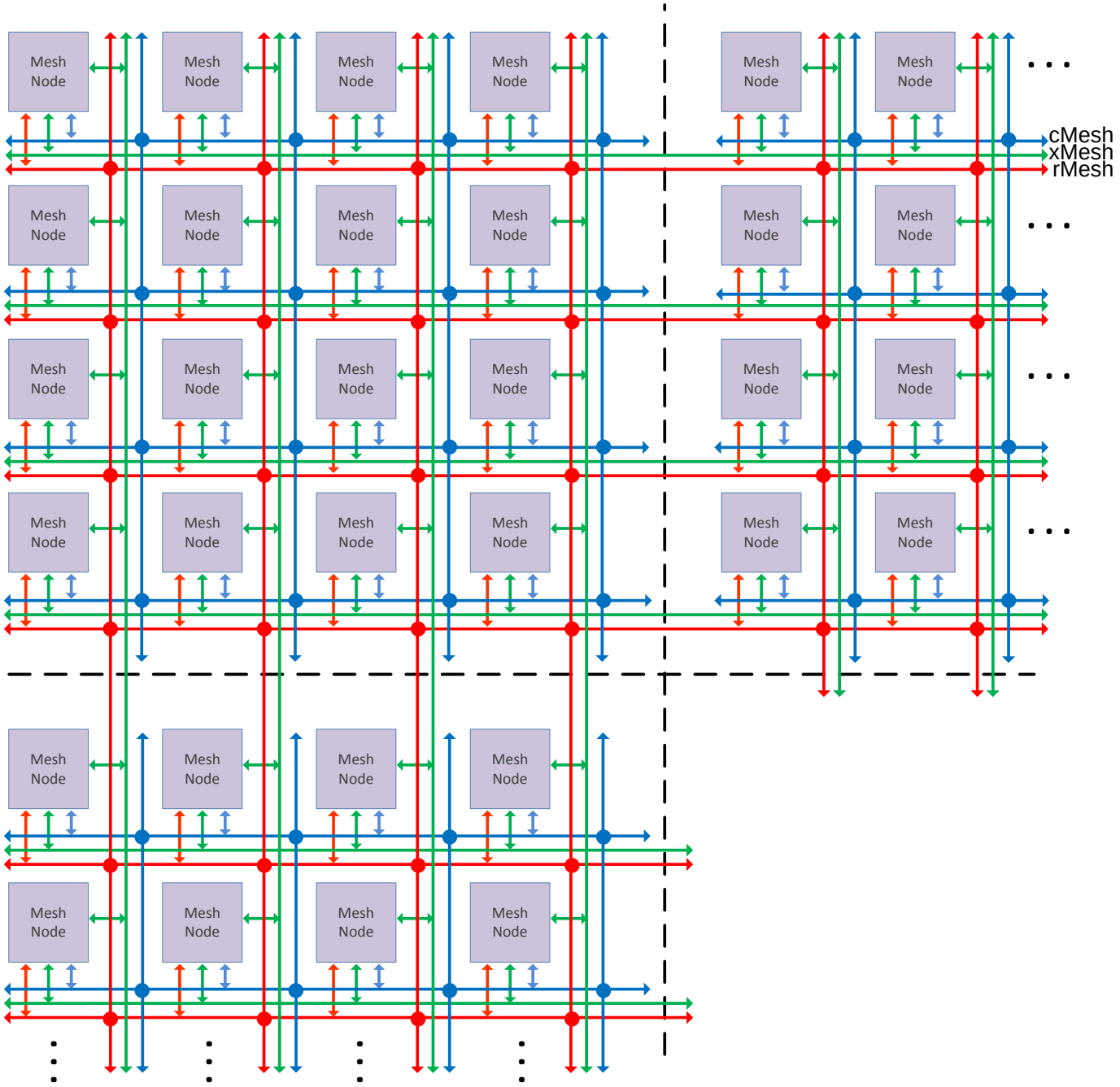
The eMesh network has a 2D mesh topology with only nearest-neighbor direct connections. Every router in the mesh is connected to the north, east, west, south, and to a mesh node. Write transactions move through the network, with a latency of 1.5 clock cycles per routing hop. A transaction traversing from the left edge to right edge of a 64- core chip would thus take 12 clock cycles. The edges of the 2D array can be connected to non-Epiphany interface modules, such as SRAM modules, FIFOs, I/O link ports, or standard buses such as AHB and AXI. Alternatively, the mesh edge connections can be left unconnected if not needed in the system.

The eMesh Network-on-Chip consists of three separate and orthogonal mesh structures, each serving different types of transaction traffic:

- *cMesh:* Used for write transactions destined for an on-chip mesh node. The cMesh network connects a mesh node to all four of its neighbors and has a maximum bidirectional throughput of 8 bytes/cycle in each of the four routing directions. At an operating frequency of 1GHz, the cMesh network has a total throughput of more than 0.5 Terabit/sec.
- *rMesh:* Used for all read requests. The rMesh network connects a mesh node to all four of its neighbors and has a maximum throughput of 1 read transaction every 8 clock cycles in each routing direction. .
- *xMesh:* Used for write transactions destined for off-chip resources and for passing through transactions destined for another chip in a multi-chip system configuration. The xMesh network allows an array of chips to be connected in a mesh structure without glue logic. The xMesh network is split into a south-north network and an east-west network. The maximum throughput of the mesh depends on the available-off chip I/O bandwidth. Current silicon versions of the Epiphany architecture can sustain a total off-chip bandwidth of 8GB/sec.

Figure 7 shows a connection diagram of the three mesh networks. The example shows an Epiphany chip configuration with 16 mesh nodes per chip. Each mesh node is connected to all three mesh networks. The only difference between larger-array chips and smaller-array chips is in the divisor placement between the on-chip and off-chip transaction routing model. The off-chip boundary is indicated by a dotted line in the figure.

Figure 7: eMesh™ Network Topology



The cMesh on-chip network has significantly lower latency and higher bandwidth than the xMesh network, so tasks with significant inter-task communication should be placed together on the same chip for optimal performance.

Key features of the eMesh network include:

- *Optimization of Write Transactions over Read Transactions*. Writes are approximately 16x more efficient than reads for on-chip transactions. Programs should use the high write-transaction bandwidth and minimize inter-node, on-chip read transactions.
- *Separation of On-Chip and Off-Chip Traffic*. The separation of the xMesh and cMesh networks decouples off-chip and on-chip communication, making it possible to write on-chip applications that have deterministic execution times regardless of the types of applications running on neighboring nodes.
- *Deadlock-Free Operation*. The separation of read and write meshes—together with a fixed routing scheme of moving transactions first along rows, then along columns—guarantees that the network is free of deadlocks for all traffic conditions.
- *Scalability*. The implementation of the eMesh network allows it to scale to very large arrays. The only limitation is the size of the address space. For example, a 32-bit Epiphany architecture allows for building shared memory systems with 4,096 processors and a 64-bit architecture allows for scaling up to 18 billion processing elements in a shared memory system.

5.2 Routing Protocol

The upper 12 bits of the destination address are used to route transactions to their destination. Along the way, these upper bits—6 bits for row and 6 bits for column—are compared to the row-column ID of each mesh node in the routing path. Transactions are routed east if the destination-address column tag is less than the column ID of the current router node, and they are routed west if the destination-address column tag is greater than the column ID of the current router node.

Transactions first complete routing along a single row before traveling in a column direction. When the destination-address column tag matches the mesh-node column ID, a similar comparison is made in the row direction to determine whether the transaction should be routed to the south or to the north. The transaction routing continues until both the row tag and column tag for the destination match the row and column ID of the current mesh node. Then, the transaction is routed into the network interface of mesh node.

Table 3 summarizes the routing rules for the combinations of mesh-node IDs and transaction addresses.

Table 3: Routing Protocol Summary

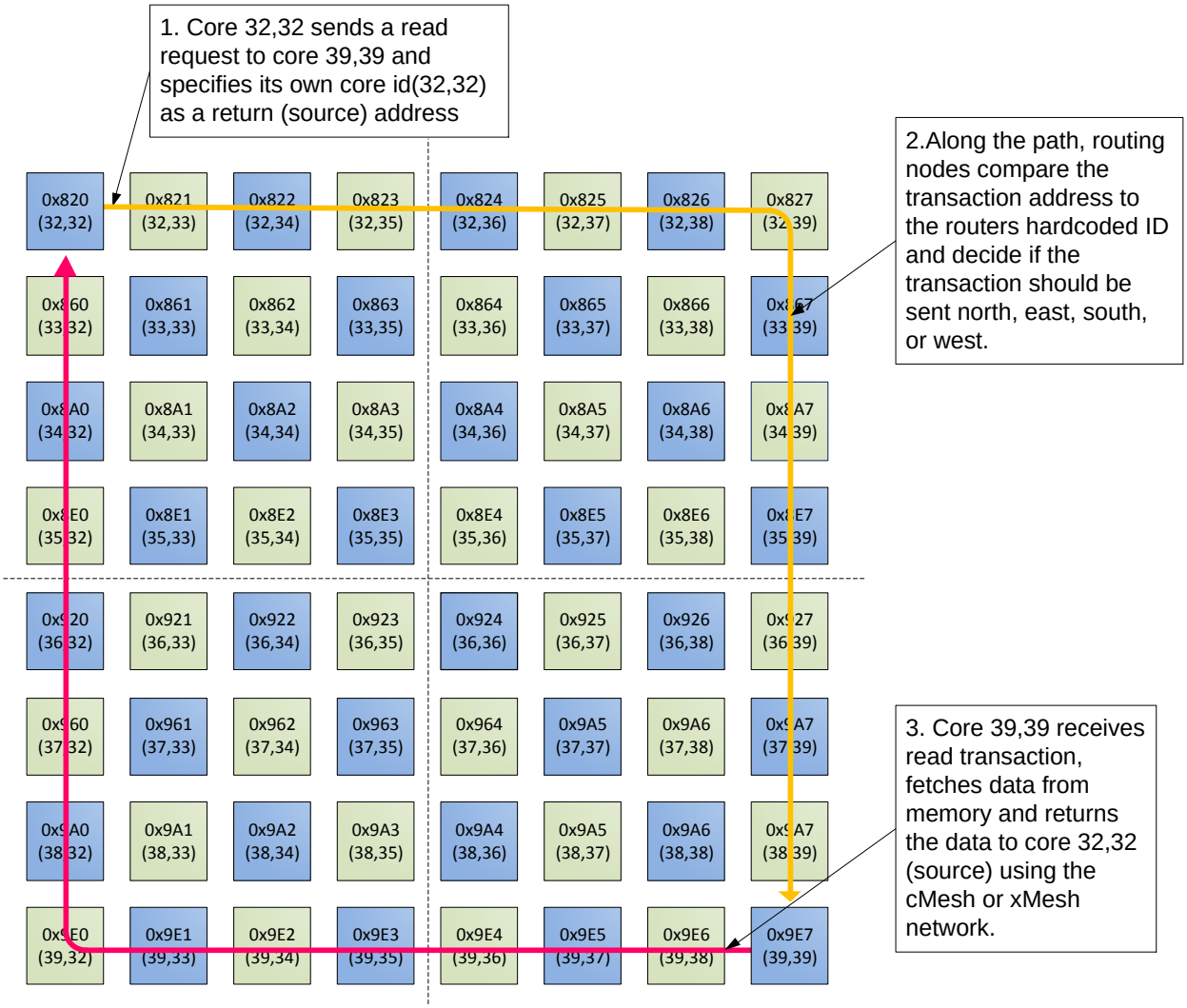
Address-Row Tag	Address-Column Tag	Routing Direction
Greater Than Mesh-Node Column	Don't Care	East
Less Than Mesh-Node Column	Don't Care	West
Matches Mesh-Node Column	Less Than Mesh-Node Row	North
Matches Mesh-Node Column	Greater Than Mesh-Node Row	South
Matches Mesh-Node Column	Matches Mesh-Node Row	Into Mesh Node

5.3 Read Transactions

Read transactions are non-blocking and are initiated as posted read requests to an address within the mesh fabric. A read request is sent out on the rMesh network and propagates towards the mesh node or external resource using the routing algorithm described in the previous section.

The source address is sent along with the read transaction on the outgoing read request. After the data has been read from the read address, the data is returned to the source address on the cMesh or xMesh network. The process is completely non-blocking, allowing transparent interleaving of read transactions from all mesh nodes. Figure 8 shows an example.

Figure 8: eMesh™ Routing Example



5.4 Direct Inter-Core Communication

Figure 9 shows how the shared-memory architecture and the eMesh network work productively together. In the example, a dot-product routine writes its result to a memory location in another mesh node. The only thing required to pass data from one node to another is the setting of a pointer. The hardware decodes the transaction and determines whether it belongs to the local node's memory or to another node's memory. Since the on-chip cMesh network can accept write transactions at the same rate that a processor core can dispatch them, the example runs without pipeline stalls, despite executing a node-to-node write in the middle of the program stream. Using this method, programmers can reduce the cost of write-based inter-node communication to zero.

Figure 9: Pointer Manipulation Example

C-CODE		ASSEMBLY

`//VecA array at 0x82002000`
`//VecB array at 0x82004000`
`//remote_res at 0x92004000`
 	
`for (i=0; i<100; i++){`
`  loc_sum+=vecA[i]*vecB[i];`
`}`
`remote_res=loc_sum;`
	→	`//R0=pointer to VecA`
		`//R2=pointer to VecB`
		`//R6=pointer to remote_res`
		`//R4=loc_sum;`
		
		`   MOV   R5,#100;`
		`_L:  LDR   R1,[R0],#1;`
		`   LDR    R3,[R2],#1;`
		`   FMADD  R4,R1,R3;`
		`   SUB    R5,R5,#1;`
		`   BNE    _L;`
		`   STR          R4,[R6];`

5.5 Arbitration Scheme

The routers at every node in all three mesh networks contain round-robin arbiters. The arbitration hardware, in combination with the routing topologies, ensures that there are no deadlocks. The round-robin scheme also ensures that there is some split of available bandwidth between the competing agents on the network. The large on-chip bandwidth and non-blocking nature of the write network guarantees that no agent needs to wait more than a few clock cycles for access to the mesh. Applications requiring exact and deterministic bandwidth can implement network-resource interleaving in software.

5.6 Data Sizes and Alignment

The eMesh network supports byte, half-word, word, or double-word atomic transactions. Mesh data is always aligned to the least-significant bits (LSBs). Maximum bandwidth is obtained with double word transactions. All transactions should have addresses aligned according to the transaction data size.

5.7 Multicast Routing

The eMesh supports efficient broadcasting of data to multiple cores through a special "multicast" routing mode. To use the multicast routing method, set the CTRLMODE field in the CONFIG register to 0011 in the master core sending eMesh write transactions. In multicast mode, the normal eMesh routing algorithm described in 5.2 is overridden and the transaction is instead routed radially outwards from the transmitting node. The write destination address is compared to the value found in the MULTICAST register at each eMesh node. If the eMesh write transaction destination address matches the MULTICAST register, the transaction enters the node.

5.8 Detour Routing Support

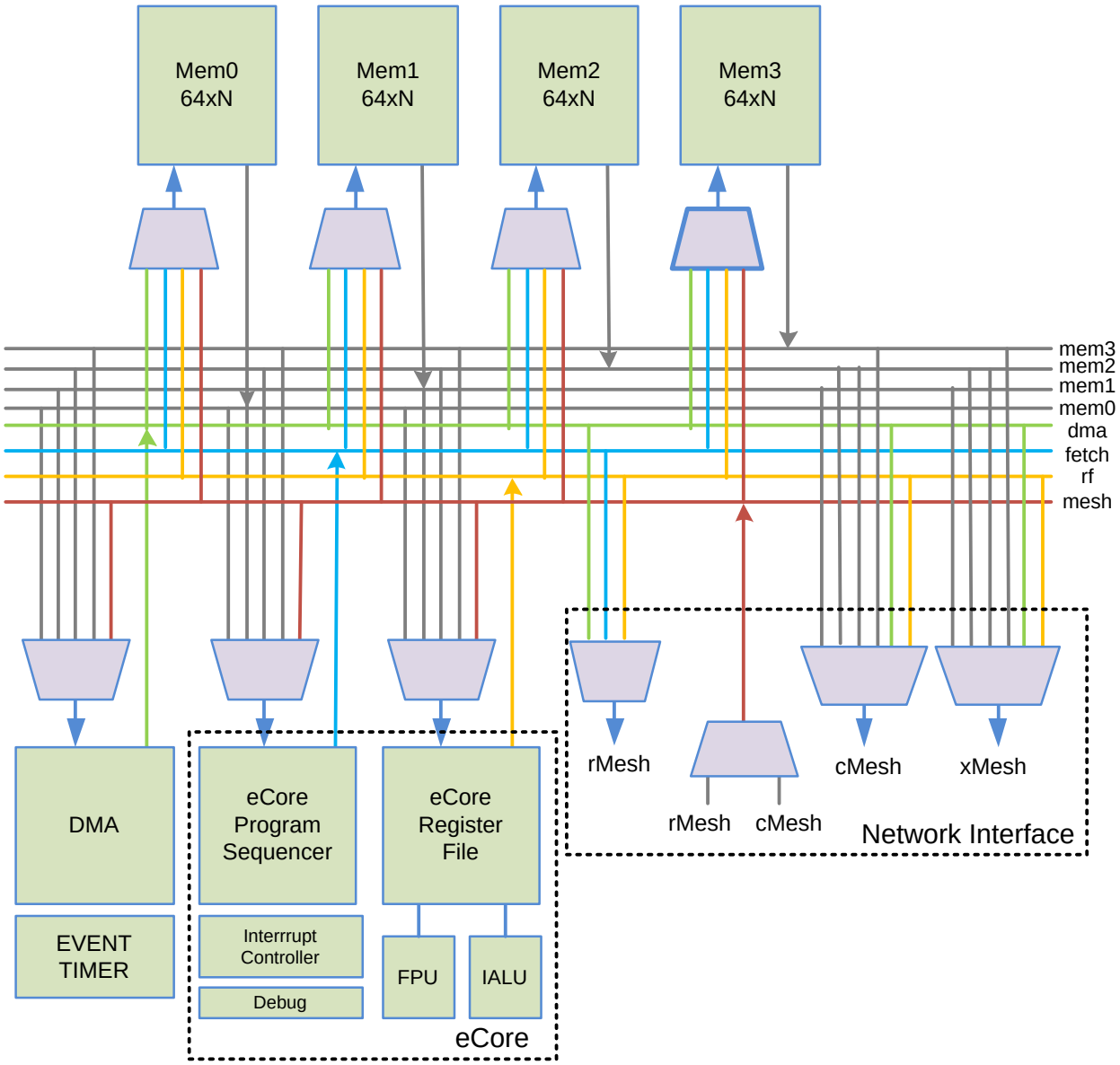
The Epiphany-IV support modifying the eMesh routing at each node on a per mesh basis using the CMESHROUTE, XMESHROUTE, and RMESHSROUTE registers. Using these registers, it's possible to block or reroute transactions.

6 Processor Node Subsystem

6.1 Processor Node Overview

Figure 10 shows the components at each processor node, which include: an eCore RISC CPU, multi-bank local memory, multicore-optimized DMA engine, event monitor, and network interface. The node connects to the Epiphany eMesh network through the network interface, a single point of access.

Figure 10: Processor Node Overview



eCore CPU

The heart of each processor node is the eCore CPU, a floating-point RISC microprocessor designed specifically for multicore processing and tuned to achieve a balance between performance, energy efficiency, and ease-of-use for many real-time applications. This balance of performance and data throughput makes performance levels close to 2 GFLOPS attainable in a large number of signal-processing kernels.

Local Memory

A local memory system supports simultaneous instruction fetching, data fetching, and multicore communication. To accomplish this, the local memory is divided into four 8-byte-wide banks, each 8KB in size.

On every clock cycle, the following operations can occur:

- 64 bits of instructions can be fetched from memory to the program sequencer.
- 64 bits of data can be passed between the local memory and the CPU's register file.
- 64 bits can be written into the local memory from the network interface.
- 64 bits can be transferred from the local memory to the network using the local DMA.

In aggregate, the local memory supports 32 bytes of memory bandwidth per clock cycle, equivalent to 32 GB /sec at 1GHz. To maximize bandwidth, use doubleword transactions and place data and instructions so that two masters never access the same memory bank simultaneously.

Direct Memory Access (DMA) Engine

The DMA engine accelerates data movement between processor nodes within the eMesh fabric. The engine was custom designed for the eMesh fabric and operates at the same speed as the eMesh, allowing it to generate a double word transaction on every clock cycle.

Event Timers

Each processor node has two 32-bit event timers that can operate independently to monitor key events within the processor node. The timers can be used for program debug, program optimization, load balancing, traffic balancing, timeout counting, watchdog timing, system time, and numerous other purposes.

Network Interface

The network interface connects all other parts of the processor node to the eMesh network-on-chip. The network interface decodes load and store instructions, program counter addresses, and DMA-transaction addresses. It also decides whether a transaction is destined for the processor node itself (in which case bits [31:20] of the address are all zero) or for the mesh network. Arbitration is performed if more than one transaction is going out, in the same clock cycle, on one of the three network meshes. The network operates at the same frequency as the CPU and can output one transaction on the network per clock cycle. For double word write transactions, 8 bytes can be pushed onto the network on every clock cycle without stalling the pipeline.

Memory Protection Unit

The memory protection unit provides the ability guard against unintended access of specific memory regions or cores.

6.2 Mesh-Node Crossbar Switch

The local memory in a processor node is split into 4 banks that are 8 bytes wide. The banks can be accessed in 1-byte transfers or in 8-byte transfers. All banks can be accessed once per clock cycle and operate at the same frequency as the CPU. The memory system in a single processor node thus supports 32GB/sec memory bandwidth at an operating frequency of 1 GHz.

Four masters can access the processor node local memory simultaneously:

- **Instruction Fetch:** This master fetches one 8-byte instruction from local memory into the instruction decoder of the program sequencer. The CPU's maximum instruction issue rate is two 32-bit instructions per clock cycle, so in heavily loaded program conditions, the program sequencer can access a memory bank on every clock cycle. The instruction-fetch logic can also fetch instructions directly from external memory or from other cores within the Epiphany fabric.
- **Load/Store:** This master copies data between the register file and a memory bank or external memory. Loads and stores can occur as 8-, 16-, 32-, or 64-bit transfers.
- **DMA:** Once set up, a DMA channel can work completely independently from the node's CPU to move data in and out of the node. Valid data-transfer sizes are 8, 16, 32, or 64 bits.
- **External**: An external agent may access the local memory to read or write certain address locations. Also, whenever the node initiates a read from an external memory location, the

transaction comes back as a write transaction that cannot be differentiated from an externally generated transaction.

The figures below show examples of maximizing memory bandwidth by assigning data and code to memory banks within the 32 KB local memory. Figure 11 shows a program memory layout optimized for memory size.

Figure 12 shows a program memory layout using a ping-pong configuration that is optimized for program speed.

Figure 11: Program Memory Layout Optimized for Size

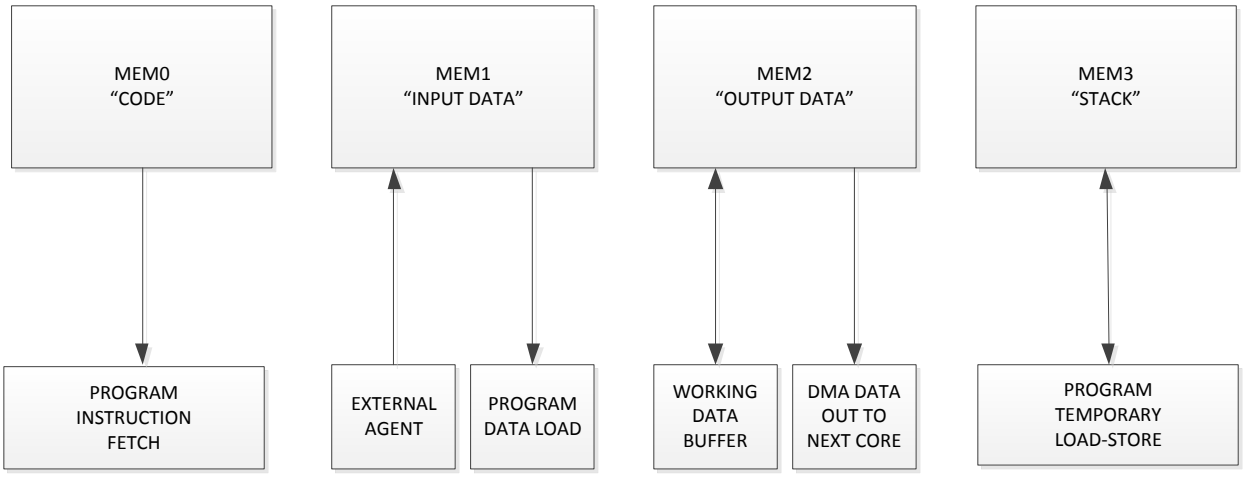
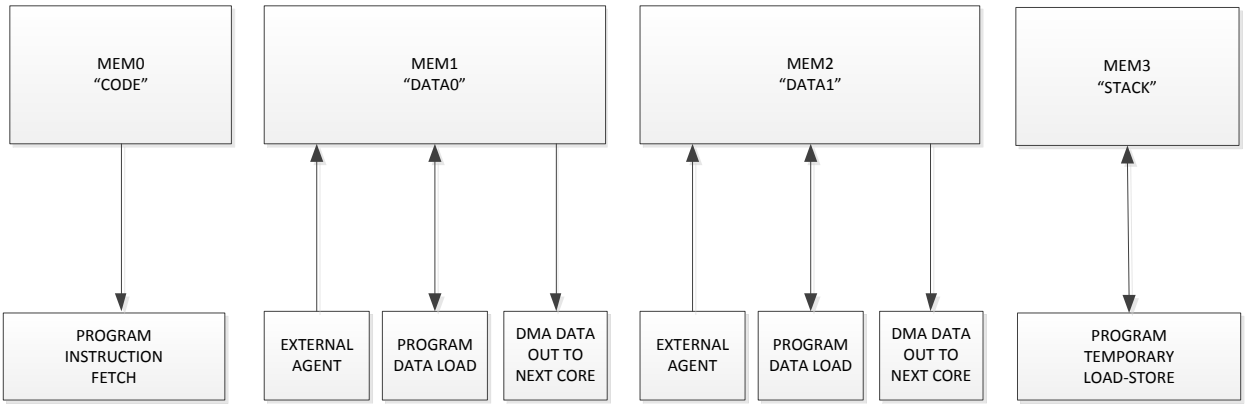


Figure 12: Program Memory Layout Optimized for Speed



6.3 Mesh-Node Arbitration

The crossbar switch within a processor node implements fixed-priority arbitration. The arbiter is needed whenever there is a potential for a shared-resource conflict. Table 4 illustrates the access priority for the different masters within the processor node for different shared resources.

Table 4: Processor node Access Priorities

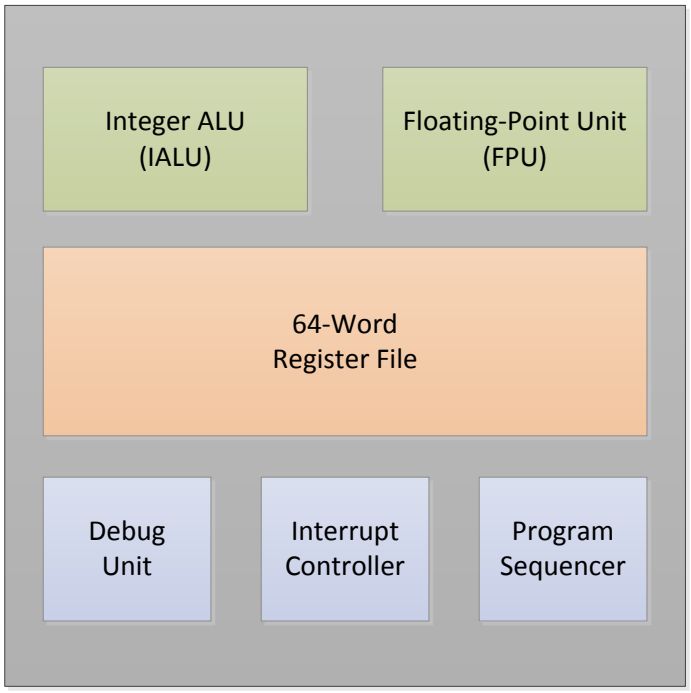
Shared Resource	Priority #1	Priority #2	Priority #3	Priority #4	Priority #5
Mem0	cMesh	rMesh	Load-Store	Program Fetch	DMA
Mem1	cMesh	rMesh	Load-Store	Program Fetch	DMA
Mem2	cMesh	rMesh	Load-Store	Program Fetch	DMA
Mem3	cMesh	rMesh	Load-Store	Program Fetch	DMA
rMesh	Load-Store	Program Fetch	DMA	n/a	n/a
cMesh	rMesh	Load-Store	DMA	n/a	n/a
xMesh	rMesh	Load-Store	DMA	n/a	n/a
Register File	cMesh	rMesh	Load-Store	n/a	n/a

7 eCore CPU

7.1 Overview

The different sub components of the eCore CPU are illustrated in Figure 13. The processor includes a general purpose program sequencer, large general purpose register file, integer ALU (IALU), floating point unit (FPU), debug unit, and interrupt controller.

Figure 13: eCore CPU Overview



Program Sequencer

The program sequencer supports all standard program flows for a general-purpose CPU, including:

- *Loops*: One sequence of instructions is executed several times. Loops are implemented using general-purpose branching instructions, in which case the branching can be done by label or by register.
- *Functions*: The processor temporarily interrupts the sequential flow to execute instructions from another part of the program. The CPU supports all C-function calls, including recursive functions.

- *Jumps*: Program flow is permanently transferred to another part of the program. A jump by register instruction allows program flow to be transferred to any memory location in the 32-bit address space that contains valid program code.
- *Interrupts*: Interrupt servicing is handled by the interrupt controller, which redirects the program sequencer to an interrupt handler at a fixed address associated with the specific interrupt event. Before entering the interrupt service routine, the old value of the program counter is stored so that it can be retrieved later when the interrupt service routine finishes.
- *Idle*: A special instruction that puts the CPU into a low-power state waiting for an interrupt event to return the CPU to normal execution. This idle mode is useful, for example, in signal-processing applications that are real-time and data-driven.
- *Linear:* In linear program flows, the program sequencer continuously fetches instructions from memory to ensure that the processor pipeline is fed with a stream of instructions without stalling.

Register File

The 9-port 64-word register file provides operands for the IALU and FPU and serves as a temporary power-efficient storage place instead of memory. Arithmetic instructions have direct access to the register file but not to memory. Movement of data between memory and the register file is done through load and store instructions. Having a large number of registers allows more temporary variables to be kept in local storage, thus reducing the number of memory read and write operations. The flat register file allows user to balance resources between floating-point and integer ALU instructions as any one of the 64 registers be used by the floating-point unit or IALU, without restrictions.

In every cycle, the register file can simultaneously perform the following operations:

- Three 32-bit floating-point operands can be read and one 32-bit result written by FPU.
- Two 32-bit integer operands can be read and one 32-bit result written by IALU.
- A 64-bit doubleword can be written or read using a load/store instruction.

Integer ALU

The Integer ALU (IALU) performs a single 32-bit integer operation per clock cycle. The operations that can be performed are: data load/store, addition, subtraction, logical shift,

arithmetic shift, and bitwise operations such as XOR and OR. The IALU's single-cycle execution means the compiler or programmer can schedule integer code without worrying about data-dependency stalls. All IALU operations can be performed in parallel with floating-point operations as long as there are no register-use conflicts between the two instructions. Pre- and post-modify addressing and doubleword load/store capability enables efficient loading and storing of large data arrays.

Floating-Point Unit

The floating-point unit (FPU) complies with the single precision floating point IEEE754 standard, executes one floating-point instruction per clock cycle, supports round-to-nearest even and round-to-zero rounding modes, and supports floating-point exception handling. The operations performed are: addition, subtraction, fused multiply-add, fused multiply-subtract, fixed-to-float conversion, absolute, float-to-fixed conversion.

Operands are read from the 64-entry register file and are written back to the register file at the end of the operation. No restrictions are placed on register usage. Regular floating-point operations such as floating-point multiply/add read two 32-bit registers and produce a 32-bit result. A fused multiply-add instruction takes three input operands and produces a single accumulated result. A large number of floating-point signal-processing algorithms use the multiply-accumulate operations, and for these applications the fused operations has the potential of reducing the number clock cycles significantly.

Interrupt Controller

The interrupt controller supports up to 10 interrupts and exceptions, with full support for nested interrupts and interrupt masking.

Hardware Loops

Efficient zero overhead loops are supported through built in hardware support.

Debug Unit

The debug unit provide multicore debug capabilities such as: single stepping, breakpoints, halt, and resume. For a complete description of supported debug features, please refer to the *Epiphany SDK Reference Manual.*

7.2 Data Types

The CPU architecture supports the following integer data types:

- Byte: 8 bits
- Half-Word: 16 bits (must be aligned on 2 byte boundary in memory)
- Word: 32 bits (must be aligned on 4 byte boundary in memory)
- Double: 64 bits (must be aligned on 8 byte boundary in memory)

The data types can be of signed or unsigned format, as shown below. All register-register operations operate on word types only, but data can be stored in memory as any size. For example, an array of bytes can be stored in memory by an external host, read into the register file using the byte load instruction, operated on as 32-bit integers, and then can stored back into memory using the byte store instruction.

Signed Integer Representation

msb *lsb*

$-a_{N-1}\cdot 2^{N-1}\ a_{N-2}\cdot 2^{N-2}\ a_{N-3}\cdot 2^{N-3}\ a_{N-4}\cdot 2^{N-4}\ a_{N-5}\cdot 2^{N-5}\ \ldots\ .\ .\ .\ .\ .\ a_0\cdot 2^0$

Unsigned Integer Representation

msb *lsb*

$a_{N-1}\cdot 2^{N-1}\ a_{N-2}\cdot 2^{N-2}\ a_{N-3}\cdot 2^{N-3}\ a_{N-4}\cdot 2^{N-4}\ a_{N-5}\cdot 2^{N-5}\ \ldots\ .\ .\ .\ .\ .\ a_0\cdot 2^0$

Floating-Point Data Types

The FPU supports the IEEE754 32-bit single-precision floating-point data format, shown below:

SIGN	EXP[7:0]	MANTISSA[22:0]

A number in this floating-point format consists of a sign bit, s, a 24-bit mantissa, and an 8-bit unsigned-magnitude exponent, e. For normalized numbers, the mantissa consists of a 23-bit fraction, f, and a hidden bit of 1 that is implicitly presumed to precede f22 in the mantissa. The binary point is presumed to lie between this hidden bit and f22. The least-significant bit (LSB) of the fraction is f0; the LSB of the exponent is e0. The hidden bit effectively increases the

precision of the floating-point mantissa to 24 bits from the 23 bits actually stored in the data format. This bit also ensures that the mantissa of any number in the IEEE normalized number format is always greater than or equal to 1 and less than 2. The exponent, e, can range between 1 ≤ e ≤ 254 for normal numbers in the single-precision format. This exponent is biased by +127 (254/2). To calculate the true unbiased exponent, 127 must be subtracted from e.

The IEEE standard also provides for several special data types in the single-precision floating-point format, including:

- An exponent value of 255 (all ones) with a nonzero fraction is a not-a-number (NAN). NANs are usually used as flags for data flow control, for the values of uninitialized variables, and for the results of invalid operations such as $0 * \infty$.
- Infinity is represented as an exponent of 255 and a zero fraction. Because the number is signed, both positive and negative infinity can be represented.
- Zero is represented by a zero exponent and a zero fraction. As with infinity, both positive zero and negative zero can be represented. The IEEE single-precision floating-point data types supported by the processor and their interpretations are summarized in Table 5.

Table 5: IEEE Single-Precision Floating-Point Data Types

Type	Sign	Exponent	Mantissa	Value
NAN	X	255	Nonzero	Undefined
Infinity	S	255	Zero	$(-1)^S$ * Infinity
Normal	S	1 <= e <=254	Any	$(-1)^S * (1.M_{22-0}) 2^{e-127}$
Denormal	S	0	Any	$(-1)^S$ * Zero
Zero	S	0	0	$(-1)^S$ * Zero

The CPU is compatible with the IEEE-754 single-precision format, with the following exceptions:

- No support for inexact flags.
- NAN inputs generate an invalid exception and return a quiet NAN. When one or both of the inputs are NANs, the sign bit of the operation is set as an XOR of the signs of the input sign bits.
- Denormal operands are flushed to zero when input to a computation unit and do not generate an underflow exception. Any denormal or underflow result from an arithmetic operation is flushed to zero and an underflow exception is generated.
- Round-to-±infinity is not supported.

By default, the FPU performs round-to-nearest even IEEE754 floating-point rounding. In this rounding mode, the intermediate result is rounded to the nearest complete number that fits within the final 32-bit floating-point data format. If the result before rounding is exactly halfway between two numbers in the destination format (differing by an LSB), the rounded result is that number which has an LSB equal to zero. Statistically, rounding up occurs as often as rounding down, so there is no large sample bias.

The FPU supports truncation rounding when the rounding mode bit is set in the Core Configuration Register. In truncate rounding mode, the intermediate mantissa result bits that are not within the first 23 bits are ignored. Over a large number of accumulations, there can be a large sample bias in the computation, so truncation rounding mode should be avoided for most applications.

The FPU detects overflow, underflow, and invalid conditions during computations. If one of these conditions is detected, a software exception signal is sent to the interrupt controller to start an exception handling routine.

Double-precision floating-point arithmetic is emulated using software libraries and should be avoided if performance considerations outweigh the need for additional precision.

7.3 Local Memory Map

Table 6 summarizes the memory map of the eCore CPU local memory.

Table 6: eCore Local Memory Map Summary

Name	Start Address	End Address	Size (Bytes)	Comment
Interrupt Vector Table	0x00	0x3F	64	Local Memory
Bank 0	0x40	0x1FFF	8KB-64	Local Memory Bank
Bank 1	0x2000	0x3FFF	8KB	Local Memory Bank
Bank 2	0x4000	0x5FFF	8KB	Local Memory Bank
Bank 3	0x6000	0x7FFF	8KB	Local Memory Bank
Reserved	0x8000	0xEFFFF	n/a	Reserved for future memory expansion
Memory Mapped Registers	0xF0000	0xF07FF	2048	Memory mapped register access
Reserved	0xF0800	0xFFFFF	n/a	N/A

All registers are memory-mapped and can be accessed by external agents through a read or write of the memory address mapped to that register or through a program executing MOVTS/MOVFS instructions. A complete listing of all registers and their corresponding addresses can be found in Appendix B. The eCore complete local memory space is accessible by any master within an Epiphany system by adding 12-bit processor node ID offset to the local address locations. Reading directly from the general-purpose registers by an external agent is not supported while the CPU is active. Unmapped bits and reserved bits within defined memory-mapped registers should be written with zeros if not otherwise specified.

7.4 General Purpose Registers

The CPU has a general-purpose register file containing 64 registers shown in Table 7. General-purpose registers have no restrictions on usage and can be used by all instructions in the Epiphany instruction-set architecture. The only general purpose register written implicitly by an

instruction is register R14, used to save a PC on a functional call. The register convention shown in Table 9 shows the register usage assumed by the compiler to ensure safe design and interoperability between different libraries written in C and or assembly. The registers do not have default values.

Table 7: General-Purpose Registers

Name	Synonym	Role in the Procedure Call Standard	Saved By
R0	A1	Argument/result/scratch register #1	Caller saved
R1	A2	Argument/result/scratch register #2	Caller saved
R2	A3	Argument/result/scratch register #3	Caller saved
R3	A4	Argument/result/scratch register #4	Caller saved
R4	V1	Register variable #1	**Callee Saved**
R5	V2	Register variable #2	**Callee Saved**
R6	V3	Register variable #3	**Callee Saved**
R7	V4	Register variable #4	**Callee Saved**
R8	V5	Register variable #5	**Callee Saved**
R9	V6/SB	Register variable #6/Static base	**Callee Saved**
R10	V7/SL	Register Variable #7/Stack limit	**Callee Saved**
R11	V8/FP	Variable Register #8/Frame Pointer	**Callee Saved**
R12	-	Intra-procedure call scratch register	Caller saved

R13	SP	Stack Pointer	N/A
R14	LR	Link Register	**Callee Saved**
R15		General Use	**Callee Saved**
R16-R27		General use	Caller saved
R28-R31		Reserved for constants	N/A
R32-R43		General use	**Callee saved**
R44-R63		General Use	Caller saved

The first four registers, R0-R3 (or A1-A4), are used to pass arguments into a subroutine and to return a result from a function. They can also be used to hold intermediate values within a function.

The registers R4-R8, R10, R11 (or V1-V5, V7-V8) are used to hold the values of a routine's local variables. The following registers are set implicitly by certain instructions and architecture convention dictates that they have fixed use. For more information regarding register usage, please refer to the Epiphany SDK reference manual.

- **Stack Pointer:** Register R13 is a dedicated as a stack pointer (SP).

- **Link Register:** The link register, LR (or R14), is automatically written by the CPU when the BL or JALR instruction is used. The register is automatically read by the CPU when the RTS instruction is used. After the linked register has been saved onto the stack, the register can be used as a temporary storage register.

7.5 Status Flags

The Core Status Register flags updated by the different instructions include:

ACTIVE

When set, it indicates that core is currently active. The core is inactive at reset and is activated by an external agent asserting an interrupt. Once activated, the core stays active until the user asserts the IDLE instruction, at which time the core enters a standby state. During the standby state, core clocks are disabled and the power consumption is minimized. Applications that need minimal power consumption should use the IDLE instruction to put the core in a standby state and use interrupts to activate the core when needed.

GID

When set it indicates that all external interrupts are blocked. The bit is set immediately on an interrupt occurring, giving the interrupt service routine enough time to save critical registers before another higher priority interrupt can occur. The flag is cleared by executing an RTI instruction, indicating the end of the service routine or by a GIE instruction indicating it is safe to allow a higher priority to begin if one is currently latched in the ILAT register.

WAND

A multicore flag set by the WAND instruction. The WAND flag is an output of the core that gets "ANDed" together with the WAND flags from other cores to produce a global WAND interrupt when cores have raised their respective flags.

AZ

The AZ (integer zero) flag set by an integer instruction when all bits of the result are zero and cleared for all other bit patterns. The flag is unaffected by all non-integer instructions.

AN

The AN (integer negative) flag set to on by an integer instruction when the most-significant bit (MSB) of the result is 1 and cleared when the MSB of the result is 0. The flag is unaffected by all non-integer instructions.

AC

The AC (integer carry) flag is the carry out of an ADD or SUB instruction, is cleared by all other integer instructions, and is unaffected by all non-integer instructions.

AV

The AV (integer overflow) flag set by the ADD instruction when the input signs are the same and the output sign is different from the input sign or by the SUB instruction when the second operand sign is different from the first operand and the resulting sign is different from the first operand. The flag is cleared by all other integer instructions and is unaffected by all non-integer instructions.

BZ

The BZ (floating-point zero) flag is set by a floating-point instruction when the result is zero. The flag unaffected by all non-floating-point instructions.

BN

The BN (floating-point negative) flag is set by a floating-point instruction when the sign bit (MSB) of the result is set to 1. The flag unaffected by all non-floating-point instructions.

BV

The BV (floating-point overflow) flag is set by a floating-point instruction when the post rounded result overflows(unbiased exponent>127), otherwise the BV flag is cleared. The flag unaffected by all non-floating-point instructions.

AVS

Sticky integer overflow flag set when the AV flag goes high, otherwise not cleared. The flag is only affected by the ADD and SUB instructions.

BVS

Sticky floating-point overflow flag set when the BV flag goes high, otherwise not cleared. The flag is unaffected by all non-floating-point instructions.

BIS

Sticky floating-point invalid flag set by a floating-point instruction if the either of the input operand is NAN, otherwise not cleared. The flag is unaffected by all non-floating-point instructions.

BUS

Sticky floating-point underflow flag set by a floating-point instruction if the result is denormal or one of the inputs to the operation is denormal, otherwise not cleared. The flag is unaffected by all non-floating-point instructions.

EXCAUSE

A three bit field indicating the cause of a software exception. A software exception edge interrupt is generated whenever this field is non-zero. The software exception cause values differ for Epiphany-III and IV and can be found in Appendix-C.

7.6 The Epiphany Instruction Set

The Epiphany instruction-set architecture (ISA) is optimized for real-time signal processing application, but it has all the features needed to also perform well in standard control code. Instruction-set highlights include:

- Orthogonal instruction set, with no restrictions on register usage.
- Instruction set optimized for floating point computation and efficient data movement.
- Post-modify load/store instructions for efficient handling of large array structures.
- Rich set of branch conditions, with 3-cycle branch penalty on all taken branches and zero penalty on untaken branches.
- Conditional move instructions to reduce branch penalty for simple control-code structures.
- Instructions with immediate modifies for high code density and low power consumption.
- Compact and efficient floating-point instruction set.

The ISA uses a split-width instruction encoding method, which improves code density compared with standard 32-bit width encoding. All instructions are available as both 16- and 32-bit instructions, with the instruction width depending on the registers used in the operation. Any command that uses registers 0 through 7 only and does not have a large immediate constant is encoded as a 16-bit instruction. Commands that use higher-numbered registers are encoded as 32-bit instructions. This encoding is transparent to the user, but is carefully integrated with the compiler to minimize C-based code size and power consumption.

The following section summarizes the instructions available in the Epiphany ISA. A complete alphabetical listing of the ISA can be found in Appendix A.

Branch Instructions

Unrestricted branching is supported throughout the 32-bit memory map using branch instructions and register jump instructions. Branching can occur conditionally, based on the arithmetic flags set by the integer or floating-point execution unit. The following table illustrates the condition codes supported by the ISA. The architecture supports two sets of flags to allow independent conditional execution and branching of instructions based on results from two separate arithmetic units. The full set of branching conditions allows the synthesis of any high-level control comparison, including: <, <=, =, ==, !=, >=, and >. Both signed and unsigned arithmetic is supported.

Table 8: Condition Codes

Code	Function	Mnemonic	Comment
0000	Equal	BEQ	AZ
0001	Not Equal	BNE	~AZ
0010	Greater Than (Unsigned)	BGTU	~AZ & AC
0011	Greater Than or Equal (Unsigned)	BGTEU	AC
0100	Less Than or Equal (Unsigned)	BLTEU	AZ | ~AC
0101	Less Than (Unsigned)	BLTU	~AC
0110	Greater Than (Signed)	BGT	~AZ & (AV ==AN)
0111	Greater Than or Equal (Signed)	BGTE	AV == AN
1000	Less Than (Signed)	BLT	AV !=AN
1001	Less Than or Equal (Signed)	BLTE	AZ | (AV != AN)
1010	Equal (Float)	BBEQ	BZ
1011	Not Equal (Float)	BBNE	~BZ
1100	Less Than (Float)	BBLT	BN & ~BZ
1101	Less Than or Equal (Float)	BBLTE	BN | BZ
1110	Unconditional Branch	B	-
1111	Branch and Link	BL	-

Load/Store Instructions

Load and store instructions move data between the general-purpose register file and any legal memory location within the architecture, including external memory and any other eCore CPU in the system. All other instructions, such as floating-point and integer arithmetic instructions, are restricted to using registers as source and destination operands.

The ISA supports the following addressing modes:

- **Displacement Addressing:** The memory address is calculated by adding an immediate offset to a base register value. The immediate offset is limited to 3 bits for 16-bit load/store instructions or 11 bits for 32-bit load/store instructions. The base register value is not modified by the load/store operation. This mode is useful for accessing local variables.
- **Indexed Addressing:** The memory address is calculated by adding a register value offset to a base register value. The base register value is not modified by the load/store operation. This mode is useful in array addressing.
- **Post-Modify Auto-increment Addressing:** The memory address is taken directly from the base register value. After the memory operation has completed, a register offset is added to the base register value and written back to the base register. This mode is useful for processing large data arrays and for implementing an efficient stack-pop operation.

Byte, short, word, and double data types are supported by all load/store instruction formats. All loads and stores must be aligned with respect to the data size being used. Short (16-bit) data types must be aligned on 16-bit boundaries in memory, word (32-bit) data types must be aligned on 32-bit boundaries, and double (64-bit) data types must be aligned on 64-bit boundaries. Unaligned memory accesses returns unexpected data and generates a software exception. Double data-type load/store instructions must specify an even register in the general-purpose register file. The corresponding odd register is written implicitly. Attempts to use odd registers with double data format is flagged as an error by the assembler.

Integer Instructions

General-purpose integer instructions, such as ADD, SUB, ORR, AND, are useful for control code and integer math. These instructions work with immediate as well as register-based operands. The instructions update the integer status bits of the STATUS register.

Floating-Point Instructions

An orthogonal set of IEEE754-compliant floating-point instructions for signal processing applications. These instructions update the floating-point status bits of the STATUS register.

Secondary Signed Integer Instructions

The basic floating point instruction set can be substituted with a set of signed integer instructions by setting the appropriate mode bits in the CONFIG register [19:16]. These instructions use the same opcodes as the floating-point instructions and include: IADD, ISUB, IMUL, IMADD, IMSUB.

Register Move Instructions

All register moves are done as complete word (32-bit) entities. Conditional move instructions support the same set of condition codes as the branch instructions specified in Table 12.

Program Flow Instructions

A number of special instructions used by the run time environment to enable efficient interrupt handling, multicore programming, and program debug.

The following set of tables summarizes the instructions available in the ISA.

Table 9: Instruction Set Syntax

Field	Meaning
<COND>	One of 16 condition codes.
RD	Destination Register. Can be any one of the general-purpose registers.
RN	Primary Source Register. Can be any one of the general-purpose registers.
RM	Secondary Source Register. Can be any one of the general-purpose registers.
<OP2>	Secondary operand. An immediate value or secondary sources register. Register can be any one of the general-purpose registers. Legal immediate value is <simm3> for 16-bit instructions and <simm11> for 32-bit instructions.
<size>	Data size selector. Options are B,H,L,D meaning Byte, Half, Long, Double. For single word transactions the field can be left blank.
<offset>	Load-store address offset. Valid offset values are <imm3> for 16-bit instructions and <imm11> for 32-bit instructions. Offsets are scaled based on the <size> field in the load/store instruction.
<imm3>	Unsigned Immediate value with range of 0 to 7.
<imm8>	Unsigned Immediate value with range of 0 to 255.
<imm11>	Unsigned Immediate value with range of 0 to 2047.
<imm16>	Unsigned Immediate value with range of 0 to 65,535.
<simm3>	Signed immediate value with range of -4 to +3.
<simm8>	Signed immediate value with range of -128 to +127.
<simm11>	Signed immediate value with range of -1024 to +1023.
<simm24>	Signed immediate value with range of to -8,388,608 to +8,388,607.
<label>	Jump/Branch label resolved by assembler.
<instr>.l	The ".l" suffix is used to indicate a 32 bit instruction in case where both a 16 bit and 32 bit version of the same basic instruction exists.

Table 10: Branching Instructions

Instruction	Assembler	Flags	Function
Conditional Branch	*B<COND> <label>*	None	If <COND>, PC=<label>, else PC= next instr
Jump	*B <label>*	None	PC=<label>
Jump and Link	*BL <label>*	None	PC=<label>, LR=next instr.
Register Jump	*JR RN*	None	PC=RN
Register Jump and Link	*JALR RN*	None	PC=RN, LR= next instruction

Table 11: Load/Store Instructions

Instruction	Assembler	Flags	Function
Immediate Offset	*{LDR|STR}{size} RD, [RN,#+/-<offset>]*	None	(RD=[RN+/-offset, size] | [RN+/-offset, size]=RD)
Postmodify-Immediate	*{LDR|STR}{size} RD, [RN], #+/-<offset>*	None	(RD=[RN, size] | [RN, size]=RD) , RN=RN+/-offset
Register Offset	*{LDR|STR}{size} RD, [RN, +/-RM]*	None	(RD=[RN+/-RM, size] | [RN+/-RM, size]=RD)
Postmodify-Register	*{LDR|STR}{size} RD, [RN], +/-RM*	None	(RD=[RN, size] | [RN, size]=RD) , RN=RN+/-RM
Test & Set	*TESTSET RD, [RN,RM]*	None	if ([RN+RM]) { RD= ([RN+RM])} else{([RN+RM])=RD,RD =0}

Table 12: Integer Instructions

Instruction	Assembler	Flags	Function
Addition	*ADD RD,RN,<Op2>*	AN,AZ,AV, AC,AVS	RD=RN + OP2
Subtraction	*SUB RD,RN, <Op2>*	AN,AZ,AV, AC,AVS	RD=RN - OP2
Arithmetic Shift Right	*ASR RD,RN, <Op2>*	AN,AZ,AV, AC,AVS	RD=RN >>> OP2
Logical Shift Right	*LSR RD, RN, <Op2>*	AN,AZ,AV, AC,AVS	RD=RN >> OP2
Logical Shift Left	*LSL RD, RN, <Op2>*	AN,AZ,AV, AC,AVS	RD=RN << OP2
Logical Or	*ORR RD, RN, RM*	AN,AZ,AV, AC,AVS	RD= RN | RM
Logical And	*AND RD, RN, RM*	AN,AZ,AV, AC,AVS	RD= RN & RM
Logical Xor	*EOR RD, RN, RM*	AN,AZ,AV, AC,AVS	RD= RN ^ RM
Bit Reverse	*BITR RD, RN*	AN,AZ,AV, AC,AVS	RD= bit-reverse (RN)

Table 13: Floating-Point Instructions

Instruction	Assembler	Flags	Function
Floating-Point Addition	*FADD RD,RN,RM*	BN,BZ,BV, BIS,BVS,BUS	RD = RN + RM
Floating-Point Subtraction	*FSUB RD,RN,RM*	BN,BZ,BV, BIS,BVS,BUS	RD = RN - RN
Floating-Point Multiply	*FMUL RD,RN,RM*	BN,BZ,BV, BIS,BVS,BUS	RD = RN * RM
Floating-Point Multiply-Add	*FMADD RD,RN,RM*	BN,BZ,BV, BIS,BVS,BUS	RD+=RN * RM
Floating-Point Multiply-Subtract	*FMSUB RD,RN,RM*	BN,BZ,BV, BIS,BVS,BUS	RD-= RN * RM
Floating-Point Absolute	*FABS RD,RN*	BN,BZ,BV, BIS,BVS,BUS	RD = abs(RN)
Float To Fixed Point Conversion	*FIX RD,RN*	BN,BZ,BV, BIS,BVS,BUS	RD = fix(RN)
Fixed To Float Conversion	*FLOAT RD,RN*	BN,BZ,BV, BIS,BVS,BUS	RD = float(RN)

Table 14: Secondary Integer Instructions

Instruction	Assembler	Flags	Function
Signed Integer Addition	*IADD RD,RN,RM*	BN,BZ,BV, BIS,BVS,BUS	RD = RN+RM
Signed Integer Subtraction	*ISUB RD,RN,RM*	BN,BZ,BV, BIS,BVS,BUS	RD = RN-RM
Signer Integer Multiply	*IMUL RD,RN,RM*	BN,BZ,BV, BIS,BVS,BUS	RD = RN*RM
Signed Integer Multiply-Add	*IMADD RD,RN,RM*	BN,BZ,BV, BIS,BVS,BUS	RD += RN*RM
Signed Integer Multiply-Subtract	*IMSUB RD,RN,RM*	BN,BZ,BV, BIS,BVS,BUS	RD -= RN*RM

Table 15: Register Move Instructions

Instruction	Assembler	Flags	Function
Move Immediate	*MOV RD,<imm8 | imm16>*	None	RD=<imm8 | imm16>
Move Immediate (high)	*MOVT RD, <imm16>*	None	RD= RD | (<imm16> << 16)
Move Register	*MOV<COND> RD, RN*	None	If <cond>, RD=RN
Move to Special Register	*MOVTS RD, RN*	None	RD = RN
Move from Special Register	*MOVFS RD, RN*	None	RD = RN

Table 16: Program Flow Instructions

Instruction	Assembler	Flags Set	Function
Do nothing	*NOP*	None	Nothing
Idle	*IDLE*	None	Wait for interrupt
Return from subroutine	*RTS*	None	PC=LR
Return from interrupts	*RTI*	None	PC=IRET
Interrupt Disable	*GID*	None	All interrupts disabled
Interrupt Enable	*GIE*	None	All interrupts enabled
Breakpoint	*BKPT*	None	Breakpoint
Multi-breakpoint	*MBKPT*	None	Multicore breakpoint
Trap	*TRAP*	None	Halts program
Sync	*SYNC*	None	Forces an ILAT[0] on all cores in group
Wand	*WAND*	Status[3]	Multicore barrier

7.7 Pipeline Description

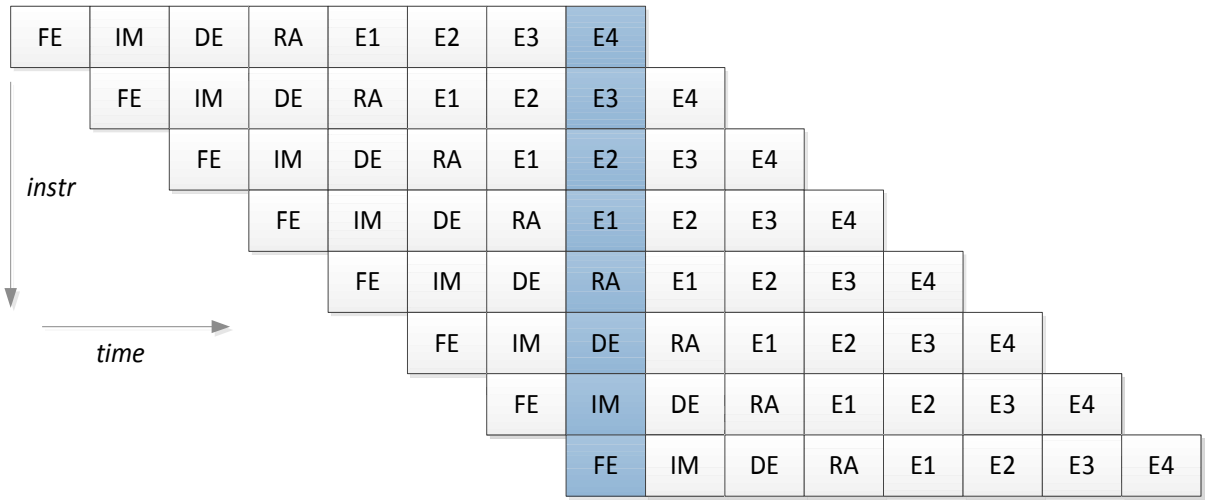
The Epiphany CPU has a variable-length instruction pipeline that depends on the type of instruction being executed. All instructions share the same instruction pipeline until the E1 pipeline stage, and instructions are guaranteed to complete once they reach that stage. Load instructions complete at stage E2, and floating-point instructions complete at stage E4. All other instructions complete at E1.

Instructions are dispatched in-order but can finish out-of-order. The pipeline controller makes sure that the integrity of the program is maintained by stalling the pipeline appropriately if there is a read-after-write (RAW) or write-after-write (WAW) pipeline hazard.

Table 17: Pipeline Stage Description

Stage	Name	Mnemonic	Action
1	Fetch Address	FE	Fetch address sent to instruction memory
2	Instruction Memory Access	IM	Instruction returns from core memory
3	Decode	DE	Instructions are decoded
4	Register Access	RA	Operands are read from register file for all instructions
5	Execution	E1	Load/store address calculation Register read from register file for memory store operation Most instructions completed Integer status flags written Branching and jumps change program flow
6	Execution	E2	Data from load instruction written to register file
7	Execution	E3	Floating-point result written to register file in case of truncation rounding mode
8	Execution	E4	Floating-point result written to register file in case of round-to-nearest-even rounding mode.

Figure 14: Pipeline Graphical View



In the execution of instructions, the CPU implements an interlocked pipeline. When an instruction executes, the target register of the read operation is marked as busy until the write has been completed. If a subsequent instruction tries to access this register before the new value is present, the pipeline will stall until the previous instruction completes. This stall guarantees that instructions that require the use of data resulting from a previous instruction do not use the previous or invalid data in the register.

Dual-Issue Scheduling Rules

The CPU has a static dual-issue architecture that allows two instructions to be executed in parallel on every clock cycle, if certain parallel-issue rules are followed. The basic requirement for dual issue is that the instruction dispatch is done in-order. This means that for two instructions to be issued in parallel (on the same clock cycle), there can be no read-after-write (RAW) or write-after-write (WAW) register dependencies between the two instructions.

For the purpose of the following data-dependency tables, the instruction set can be divided into the following instruction groups.

- IALU: ADD, SUB, ASR, LSR, LSL, EOR, AND, ORR, BITR, MOVT, MOV
- IALU2: IADD, ISUB, IMUL, IMADD, IMSUB
- FPU: FADD, FSUB, FMUL, FMADD, FMSUB, FIX, FLOAT, FABS
- LOAD/STORE: LDR, STR
- CONTROL: JR, JALR, B<COND>, BL, MOVTS, MOVFS, NOP

The following table shows the combinations of instructions that can be executed in parallel as long as there are no read or write register conflicts.

Table 18: Parallel scheduling rules

Instruction Type	IALU	FPU/IALU2	LOAD/STORE	CONTROL
IALU		YES	NO	NO
FPU/IALU2	YES		YES	NO
LOAD/STORE	NO	YES		NO
CONTROL	NO	NO	NO	

The CPU is pipelined to maximize the operating frequency, and energy efficiency and cost, of the total system. As a result of the processor pipeline, there are data-dependencies that need to be resolved by software to avoid clock cycle penalties due to processor stalls. The CPU has a fully interlocked pipeline, meaning that it automatically stalls the CPU to guarantee correct operation of sequential programs with data dependencies. The C compiler has accurate information about the CPU pipeline and is able to avoid most data dependencies in a program. However, for routines optimized in assembly, it is your responsibility to avoid data dependencies if you wish to optimize performance.

The following tables show the number of clock cycles needed to separate an instruction that is reserving a certain register and a second instruction that depends on that register. The third column in the tables gives the number of clock cycles of separation needed between the first instruction and the second instruction to avoid the stalling. It is the job of the C compiler or assembly programmer to ensure that these clock cycles can be filled with useful work. An Rz field in the table indicates that the register that does not affect any pipeline dependency table, and any register number can be used. Instruction combinations that don't have any register dependency stalls such as IALU after IALU instructions are not included in the table.

The last dependency table shows instructions that have dependencies that are fixed and are independent of instruction-to-instruction register dependencies.

Table 19: IALU Instruction Sequences

First Instruction	Second Instruction	Cycle Separation
IALU Instruction *mov rx, rz, rz*	FPU Instruction *fadd rx, rx, rx*	1

Table 20: FPU Instruction Sequences

First Instruction	Second Instruction	Cycle Separation
FPU Instruction *fadd rx, rz, rz*	FPU Instruction *fadd rx, rx, rx*	4
FPU Instruction *fadd rx, rz, rz*	Store Instruction (data) *str<size> rx, [rz, rz]*	3
FPU Instruction *fadd rx, rz, rz*	IALU Instruction *add rx, rx, rx*	4
FPU Instruction *fadd rz, rz, rz*	Branch on FPU condition *bbne _foobar;*	4
FPU Instruction *fadd rz, rz, rz*	Conditional Move *Movbne rz,rz*	4
FPU Instruction *fadd rx, rz, rz*	Special move instruction *movts CTIMER0, rx*	4

Table 21: Load Instruction Sequences

First Instruction	Second Instruction	Cycle Separation
Load Instruction *ldr<size> rx,[rz,rz]*	IALU Instruction *add rx,rx,rx*	1
Load Instruction *ldr<size> rx,[rz,rz]*	FPU Instruction *fadd rx,rx,rx*	2
Load Instruction *ldr<size> rx,[rz,rz]*	Store Instruction *str<size> rx, [rx, rx]*	1
Load Instruction *ldr<size> rx,[rz,rz]*	Load Instruction *ldr<size> rz,[rx, rx]*	1

Table 22: Stalls independent of Instruction Sequence

Instruction	Stall Cycles
External Data Load *ldr<size> rx,[rz,rz]*	10+
Byte | Half-word Internal Data Load *ldr<size> rx,[rz,rz]*	2
External Instruction Fetch	10+

Branch Penalties

The branch prediction mechanism used by the CPU assumes that the branch was not taken. There is no penalty for branches not taken. For branches that are taken, there is a three-cycle constant penalty. The table below summarizes the different branches and the penalties.

Table 23: Branch Penalties

Branch	Instruction	Penalty
Branch Not Taken	B<COND>	0
Branch Taken	B<COND>	3
Jump	B	3
Jump and Link	BL	3
Register Jump	JR	3
Register Jump and Link	JALR	3

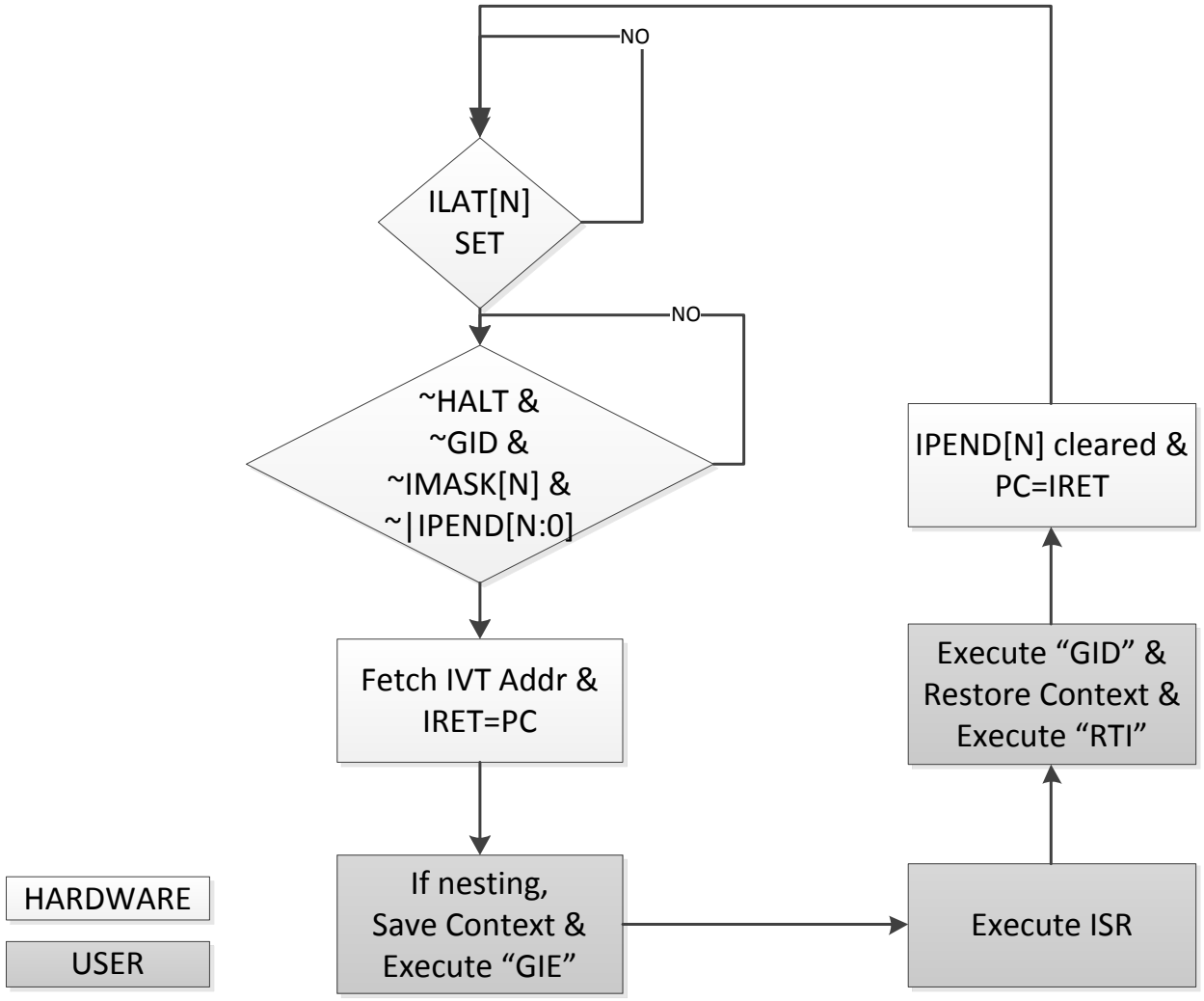
In a special case, a 1-cycle penalty occurs for jumps to 32-bit instructions that straddle two lines in the local 64-bit-wide memory. Branches with dependency on floating-point status flags also incur stall cycles if there is insufficient cycle separation between the floating-point instruction and the branching instruction.

7.8 Interrupt Controller

7.8.1 Overview

The eCore interrupt controller provides full support for prioritized nested interrupt service routines. Figure 14 shows the behavior of the hardware mechanisms within the interrupt controller and how the user can control the behavior of the system through code.

Figure 15: Interrupt Service Routine Operation



The following table summarizes the different interrupts and exceptions specified in the interrupt vector table (IVT). All interrupts are edge-based.

Table 24: Interrupt Support Summary

Interrupt/Exception	IRQ Priority	IVT Address	Event
Sync	0 (highest)	0x0	Sync hardware signal asserted
Software Exception	1	0x4	Floating-point exception, invalid instruction, alignment error
Memory Fault	2	0x8	Memory protection fault
Timer0 Interrupt	3	0xC	Timer0 has expired
Timer1 Interrupt	4	0x10	Timer1 has expired
Message	5	0x14	Message interrupt
DMA0 Interrupt	6	0x18	Local DMA channel-0 finished data transfer
DMA1 Interrupt	7	0x1C	Local DMA channel-1 finished data transfer
WAND Interrupt	8	0x20	Wired-AND signal interrupt
User Interrupt	9 (lowest)	0x24	Software generated user interrupt

The IVT entries should be populated with 32-bit relative branch instructions that point to the appropriate interrupt handlers. The Interrupt Controller uses the following registers to manage interrupts and exceptions, providing full support for nested interrupts. The priority level of the interrupt matches the actual bit position within these registers. For example, the sync interrupt with priority is mapped to bit 0 and the DMA0 interrupt is mapped to bit 6 in the ILAT register. The IRET register is a 32 bit register while the ILAT, IMASK, and IPEND are 10 bit registers.

7.8.2 Global Interrupt Disable Flag (GID)

Bit [1] of the STATUS Register is used to globally enable or disable all interrupts from affecting the core program flow. Interrupts are disabled when an interrupt starts and when the program executes a GID instruction. Interrupts are enabled again by executing an RTI or GIE instruction. The global interrupt enable bit ensures that save-and-restore functionality and context switches can be done safely, regardless of higher priority interrupts interfering with operation. All interrupts can be enabled and disabled from software using the GIE and GID instructions. The GID instruction sets the global interrupt disable bit in the STATUS register, and GIE clears the same bit. Alternatively, individual interrupts can be disabled by writing to the IMASK register.

7.8.3 User Interrupts

Individual interrupts within the ILAT register can be set and cleared from software by writing to the ILATST and ILATCL registers using the MOVTS instruction. Software interrupts can be used to degrade the interrupt priority, thereby allowing lower-priority interrupts to get handled more quickly. For example, an interrupt handler for the core timer could write to the ILATST register to force a soft interrupt with the lowest priority level, thus allowing interrupts such as the DMA interrupt, to be serviced. If the interrupt handler would stay within the timer priority handler throughout the processing, then the DMA interrupt would not get serviced until the timer interrupt had finished all the processing.

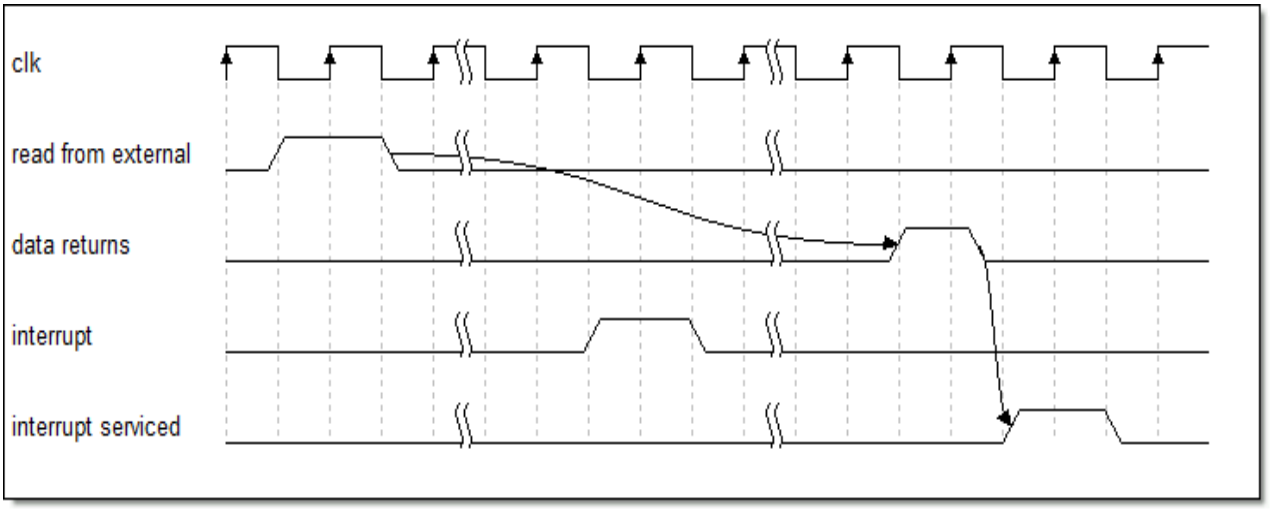
The recommended method of handling interrupt events in software is to use the interrupt registration process provided through the Epiphany run-time library. In this way, all interrupt handling can be done using standard C programming. For more details, please refer to the Epiphany SDK reference manual.

7.8.4 Interrupt Latency

To minimize interrupt latency, two requirements should be met:

- The stack should be placed in local memory. The stack is used to save and restore variables. If the stack is placed in external memory, loading of these variables could take hundreds of clock cycles.
- Loads from other cores or external memory should be minimized. Interrupts are disabled during such loads. Figure 15 illustrates the effect of external reads on interrupt service latency.

Figure 16: Interrupt Latency Dependency On External Read



7.9 Hardware Loops (LABS)

The Epiphany core supports zero overhead loops with the LC (loop counter), LS (loop start address), and LE (loop end address) registers. The three hardware loop registers must be correctly programmed before executing the critical code section. When the program counter (PC) matches the value in LE and the LC is greater than zero, the PC gets set to the address in LS. The LC register decrements automatically every time the program scheduler completes one iteration of the code loop defined by LS and LE.

The Epiphany hardware loop does place certain restrictions on the program:

- All interrupts must be disabled while inside a hardware loop.
- The start of the loop must be aligned on a double word boundary.
- The next-to-last instruction must be aligned on a double word boundary.
- All instructions in the loop set as 32 bit instructions using ".l" assembly suffix
- The minimum loop length is 8 instructions.

Example:

```
      mov  r1, %low(loop_start);
      movt r1, %high(loop_start);
      movts ls, r1;               //setting loop start address
      mov  r1, %low(loop_end);
      movt r1, %high(loop_end);
      movts le, r1;               //setting loop end address
      mov  r1, #0x10;
      movts lc, r1;               //setting loop count
      gid;                        //disabling interrupts
.balignw 8,0x01a2;                //align to 8-byte boundary
loop_start:
      add.l r1, r1, r0;           //first instruction in loop
      add.l r2, r2, r0;           //".l" forces 32 bit instruction
      add.l r3, r3, r0;
      add.l r4, r4, r0;
      add.l r5, r5, r0;
      add.l r6, r6, r0;
.balignw 8,0x01a2;                //align to 8-byte boundary
      add.l r7, r7, r0;
loop_end:
      add.l r8, r8, r0;           //last instruction
      gie;                        //enabling interrupts
```


7.10 Debug Unit

The debug unit in the Epiphany core supports the following debugging features:

- Halting and resuming operations by writing to the DEBUGCMD register from the host processor or from another Epiphany core.
- Monitoring the status of the by reading from the DEBUGSTATUS register.
- Halting the locally running program by including a BKPT instruction in the code being executed.
- Halting the locally running program and all other cores in the workgroup by including a MBKPT instruction in the code being executed.
- Single stepping code through a software implementation (see Epiphany GDB implementation)

8 Direct Memory Access (DMA)

8.1 Overview

Each Epiphany processor node contains a DMA engine to facilitate data movement across the eMesh network. The DMA engine works at the same clock frequency as the CPU and can transfer one 64-bit double word per clock cycle, enabling a sustained data transfer rate of 8GB/sec. The DMA engine has two general-purpose channels, with separate configuration for source and destination.

The main features of the DMA engine are:

- Two independent DMA channels per processor node.
- Separate specification of source/destination address configuration per descriptor and channel.
- 2D DMA operation.
- Flexible stride sizes
- DMA descriptor chaining.
- Hardware interrupts flagging to local CPU subsystem.

The following table shows the kind of transfers supported by the processor node's DMA engine.

Table 25: DMA Transfer Types

Source	Destination	Function
Local Memory	External Memory	Data read from one of the four local memory banks, and send data to the eMesh network as a write through the network interface.
External Memory	Local Memory	Read request sent to the eMesh network. You can decide if you want an interrupt indication when the last data read transaction returns (blocking DMA) or if the DMA should complete as soon as the last read request goes out on the eMesh network (non-blocking DMA).
Autodma Register	Local Memory	Write from external master. This is used when the DMA is configured in slave mode.
External Memory	External Memory	Read transaction sent to the eMesh network, destination could be anything because read transactions are split transactions. For read destinations residing outside of the Epiphany chip, care must be taken to make sure that the memory supports the split transaction routing mode needed to route the data read to the final write destination.

The DMA engine has two complete data transfer channels and supports data movement as a master as well as a slave device. In a slave configuration, the pace of the data transfers is controlled by an external master. In a master configuration, the DMA pushes a transaction every clock cycle if the necessary memory and interface resources are available.

- In the MASTER mode, the DMA generates a complete transfer transaction with a source and a destination address.
- In the SLAVE mode, the source address of a DMA configuration is ignored. The data is always taken from the DMAxAUTO register and transferred to the destination address. The pace of the transaction is driven by another master in the system, which could be an I/O device, a programmable core, or another DMA channel.

8.2 DMA Descriptors

The format of DMA descriptors is given in the table below. All descriptors should be placed in local memory and must be double-word aligned. A descriptor is brought into the DMA channel configuration register set when the startup bit is set to one or when a DMA transfer is configured in chaining mode and a new configuration should automatically be brought in at the end of a transfer.

Table 26: DMA Descriptors

Addr0+7, Addr0+6	Addr0+5, Addr0+4	Addr0+3, Addr0+2	Addr0+1, Addr0
STRIDE-INNER-DST	STRIDE-INNER-SRC	NXT_PTR	DMACONFIG
STRIDE-OUTER-DST	STRIDE-OUTER-SRC	CNT-OUTER	CNT-INNER
DST ADDRESS (HI)	DST ADDRESS (LO)	SRC ADDRESS (HI)	SRC ADDRESS (LO)

8.3 DMA Channel Arbitration

The two DMA channels have a fixed priority, with channel0 having a higher priority over channel1.

8.4 DMA Usage Restrictions

The DMA does not flag errors on incorrect usage such as: misaligned accesses or illegal memory location access. Such scenarios will cause unexpected behavior in the system and will likely result in a core or chip needing to be reset.

8.5 DMA Transfer Examples

The following example shows how to use the DMA to do a single-block transfer. To make transfers efficient, double-word transfers should be used, but in this case we are using byte transfers.

```
MOV  R1, 0x8          ; set the startup bit
MOVT R1, _1D_DESCR    ; put descriptor pointer in the upper 16 bits
MOVTS DMA0CONFIG, R1 ; start a DMA transfer by writing to the
                      ; DMA config register.

_1D_DESCR;
.word 0x00000003; configure in master mode and enable
.word 0x00010001; increment src/dst address by 1 byte each transaction
.word 0x00010008; transfer has 8 transactions in a single inner loop
.word 0x00000000; outer loop stride not used in this example
.word 0x00002000; set source address to 0x2000, a local address
.word 0x92000000; set destination address to an external address
```


9 Event Timers

The Epiphany architecture supports a distributed set of event timers that can be used to sample real-time events within the system. The type of event to monitor is controlled through the CONFIG register.

- *Clk:* General-purpose clock-cycle counter. Can be used to measure time, to profile function execution time, for real-time operating systems, and for many other purposes.
- *Idle:* Counts the number of clock cycles spent in idle. Can be used to balance the load on different CPUs.
- *IALU valid instructions:* Counts the number of IALU instructions issued.
- *FPU valid instructions:* Counts the number of FPU instructions issued.
- *Dual issues instructions:* Counts the number of cycles with two instructions issued simultaneously.
- *E1 stalls:* Counts the number of pipeline stalls due to load/store register dependencies.
- *RA stalls:* Counts all register dependency pipeline stalls.
- *Fetch contention stalls:* Counts the number of stall cycles due to memory-bank contention in the processor node. Can be used to uncover issues with program code placement.
- *Ext fetch stalls:* Counts the number of clock-cycle stalls due to the program sequencer waiting for an instruction to return from external memory. Can be used to uncover areas of the code that are running from external memory instead of local memory.
- *Ext data stalls:* Counts the number of clock-cycle stalls due to a load instruction accessing external memory and stalling the pipeline. Can be used to uncover areas of the code that are accessing variables from external memory instead of local memory.
- *Mesh traffic:* Counts wait or access events on the local cMesh network node. Mesh event monitor event selection is programmed through the MESHCONFIG registers

The CTIMER0 register contains the current value of the event being monitored. The register counts down from a high value to zero, decrementing every time the chosen event is detected. When the timer reaches zero, the counter stops counting and an interrupt is issued to the interrupt controller. The event count mode is set in the CONFIG register.

The CTIMER1 register contains the current value of the event being monitored. The register counts down from a high value to zero, decrementing every time the chosen event is detected. When the timer reaches zero, the counter stops counting and an interrupt is issued to the interrupt controller. The event count mode is set in the CONFIG register.

10 Memory Protection Unit (LABS)

The Memory Protection Unit allows the user to specify parts or all of the local memory as read only memory. The 32KB local memory is split into 8 4KB page that can be independently set to read-only. If a write is attempted to a page that has been set to read only, and the memory fault exception bit in the ILAT register is set. The MEMPROTECT register can be used to help debug program faults related to stack overflow and multicore memory clobbering.

Appendix A: Instruction Set Reference

The following section contains an alphabetical listing of the Epiphany Instruction Set.

ADD

Description: The ADD instruction adds an integer register value (RN) with a second integer operand (OP2), which can be an immediate value (SIMM3 or SIMM11) or register value (RM).

Syntax: ADD <RD>, <RN>, <RM>
ADD <RD>, <RN>, #SIMM3
ADD <RD>, <RN>, #SIMM11

<RD> Destination register
<RN> First operand register
<RM> Second operand register
<SIMM3 | SIM11> Three or eleven bit signed immediate value.

Flags Updated: AN Flag
AZ Flag
AV Flag
AC Flag

Operation: RD = RN + <OP2>
AN = RD[31]
AC = CARRY OUT
if (RD[31:0] == 0) { AZ=1 } else { AZ=0 }
if ((RD[31] & ~RM[31] & ~RN[31]) | (~RD[31] & RM[31] & RN[31]))
{ OV=1 }
else { OV=0 }
AVS = AVS | AV

Example:
```
ADD R2,R1,#2          ;
ADD R2,R1,#-100       ;
ADD R1,R1,R3          ;
```


AND

Description: The AND instruction logically "AND"s the operand in register RN with the operand in register RM and places the result in register RD.

Syntax: AND <RD>, <RN>, <RM>

<RD> Destination register
<RN> First operand register
<RM> Second operand register

Flags Updated: AN Flag
AZ Flag
AV Flag
AC Flag

Operation:
RD = RN & RM
AN = RD[31]
AV = 0
AC = 0
If (RD[31:0] == 0) { AZ=1 } else { AZ=0 }

Example:
```
AND R2,R1,R0          ;
```


ASR

Description: The ASR instruction performers an arithmetic shift right of the RN operand based on the shift value (OP2). OP2 is a 5 bit unsigned immediate value or an unsigned shift value contained within the first 5 bits of operand register RM. The result is sign extended using bit RN[31]. The result is placed in register RD.

Syntax: ASR <RD>, <RN>, <RM>
ASR <RD>, <RN>, #IMM5

<RD> Destination register
<RN> First operand register
<RM> Second operand register
<IMM5> Five bit unsigned immediate value

Flags Updated: AN Flag
AZ Flag
AV Flag
AC Flag

Operation: RD = RN >>> <OP2>
AN = RD[31]
AV = 0
AC = 0
if (RD[31:0] == 0) { AZ=1 } else { AZ=0 }

Example:
```
ASR R0,R1,R2;
```


B<COND>

Description: The branch instruction causes a branch to a target address based on the evaluation of one of 16 condition codes. The instruction allows conditional and unconditional branching forwards and backwards relative to the current value of the program counter. All branches are relative with respect to the current program counter.

Syntax: B<COND> <SIMM8>
B<COND> <SIMM24>

<COND> One of 15 conditions to evaluate before performing the jump(branch). The allowed branching opcodes are: BEQ, BNE, BGT, BGTE, BLTE, BLT, BLTU, BLTEU,BGTU,BGTEU, BBEQ, BBNE, BBLT, BBLTE. For a further description of the condition fields, refer to the condition instruction set summary section. An empty field refers to un unconditional branch.

<SIMM8> A signed immediate value to be added the current PC to create a new instruction fetch address. The value is sign extended to 32-bit and left shifted by 1 bit before being added to the PC.

<SIMM24> A signed immediate value to be added the current PC to create a new instruction fetch address. The value is sign extended to 32-bit and left shifted by 1 bit before being added to the PC.

Flags Updated: None

Operation: IF (Passed)<COND>)) then
PC = PC +(SignExtend(SIMM) <<1)

Example:

```
_inf:     MOV R0,#10  ; loop 10 times
```



```
_loopA:   ADD R1,R1#1 ; some operation
          SUB R0,R0,#1 ; decrement loop counter
          BEQ _loopA  ; branch while true
          B  _inf   ; keep executing forever
```

BL

Description: The branch instruction causes the upcoming PC to be saved in the LR register followed by a branch to a target. The branch is relative with respect to the current program counter.

Syntax: BL <SIMM8>
BL <SIMM24>

<SIMM8> A signed immediate value to be added the current PC to create a new instruction fetch address. The value is sign extended to 32-bit and left shifted by 1 bit before being added to the PC.

<SIMM24> A signed immediate value to be added the current PC to create a new instruction fetch address. The value is sign extended to 32-bit and left shifted by 1 bit before being added to the PC.

Flags Updated: None

Operation: LR=next PC;
PC = PC +(SignExtend(SIMM) <<1)

Example: `BL _MY_FUNC; save PC to LR and jump to _MY_FUNC`

Description: The BITR instruction reverses the order of the bits in the operand RN, the LSB becomes the MSB and the MSB becomes the LSB, etc. and places the result in register RD.

Syntax: BITR <RD>, <RN>

<RD> Destination register

<RN> First operand register

Flags Updated: AN Flag

AZ Flag

AV Flag

AC Flag

Operation:

```
for(i=0;i<32;i=i+1){
        RD[i]=RN[31-i];
}
if (RD[31:0]==0) { AZ=1 } else { AZ=0}
AN  = RD[31]
AV  = 0
AC  = 0
```

Example:

```
MOV  R0,%low(x87654321)  ;
MOV  R0,%high(x87654321) ;
BITR R0,R0               ;R0 gets 0x84C2A6B1
```


BKPT

Description: The BKPT instruction causes the processor to halt and wait for external inputs. The instruction is only be used by the debugging tools such as GDB and should not be user software. The instruction is included here only for the purpose of reference.

Syntax: BKPT

EOR

Description: The EOR instruction logically XORs the operand in register RN with the operand in register RM and places the result in register RD.

Syntax: EOR <RD>, <RN>, <RM>

<RD> Destination register

<RN> First operand register

<RM> Second operand register

Flags Updated: AN Flag

AZ Flag

AV Flag

AC Flag

Operation: RD = RN ^ RM

AN = RD[31]

AV = 0

AC = 0

if (RD[31:0]==0) { AZ=1 } else { AZ=0}

Example:

```
EOR R2,R0,R1    ;
```


FABS

Description: The FABS instruction calculates the absolute value of a floating-point value in register value RN and and places the result in register RD. The operation updates the floating-point arithmetic flags.

Syntax: FABS <RD>, <RN>.

<RD> Destination register.

<RN> First operand register

Flags Updated:

BN Flag

BZ Flag

BV Flag

BIS Flag

BUS Flag

BVS Flag

Operation:

RD = abs(RN)

N = RD[31]

if (RD[30:0]==0) { BZ=1 } else { BZ=0}

if (UnbiasedExponent(RD) > 127) { BV=1 } else { BV=0}

if (UnbiasedExponent(RD) < -126) { BUS=1 } else { BUS=BUS}

if (RM or RN == NAN) { BIS=1 } else { BIS=BIS}

BVS = BVS | BV;

Example:

```
FABS R2,R1;
```


FADD

Description: The FADD instruction adds two 32-bit floating-point operands together and places the result in a third register. The operation updates the floating point arithmetic flags.

Syntax: FADD <RD>, <RN>, <RM>

<RD> Destination register

<RN> First operand register

<RM> Second operand register

Flags Updated: BN Flag

BZ Flag

BV Flag

BIS Flag

BUS Flag

BVS Flag

Operation:

```
RD=RN + RM
BN = RD[31]
if (RD[30:0]==0) { BZ=1 } else { BZ=0}
if (UnbiasedExponent(RD) > 127) {B OV=1 } else { BV=0}
if (UbiasedExponent(RD) < -126) { BUS=1 } else { BUS=BUS}
if (RM or RN == NAN) { BIS=1 } else { BIS=BIS}
BVS = BVS | BV;
```

Example:

```
FADD R2,R2,R0;
```


FIX

Description: These FIX instruction converts the floating-point RN operand to a 32-bit fixed-point signed integer result. The floating-point operand is rounded or truncated. The result is placed in register RD. A NAN input returns a floating-point all ones result. All underflow results, or input which are zero or denormal, return zero. Overflow result always returns a signed saturated result: 0x7FFFFFFF for positive, and 0x80000000 for negative.

Syntax: FIX <RD>, <RN>

<RD> Result register for converted fixed point result

<RN> Floating-point operand register to convert.

Flags Updated:

BN Flag

BZ Flag

BV Flag

BIS Flag

BUS Flag

BVS Flag

Operation:

RD = fix(RN)

N = RD[31]

if (RD[30:0]==0) { BZ=1 } else { BZ=0}

if (UnbiasedExponent(RD) > 127) { BV=1 } else { BV=0}

if (UbiasedExponent(RD) < -126) { BUS=1 } else { BUS=BUS}

if (RM or RN == NAN) { BIS=1 } else { BIS=BIS}

BVS = BVS | BV;

Example:

```
FIX R2,R1;
```


FLOAT

Description: The FLOAT instructions convert the fixed-point operand in Rn to a floating-point result. The final result is placed in register Rd. Rounding is to nearest (IEEE) or by truncation, to a 32-bit boundary, as defined by the rounding mode. Overflow returns ±infinity (round-to-nearest),underflow returns ±zero.

Syntax: FLOAT <RD>, <RN>

<RD> Result register for converted fixed point result

<RN> Floating-point operand register to convert.

Flags Updated:

BN Flag

BZ Flag

BV Flag

BIS Flag

BUS Flag

BVS Flag

Operation:

RD = float(RN)

N = RD[31]

if (RD[30:0]==0) { BZ=1 } else { BZ=0}

if (UnbiasedExponent(RD) > 127) { BV=1 } else { BV=0}

if (UbiasedExponent(RD) < -126) { BUS=1 } else { BUS=BUS}

if (RM or RN == NAN) { BIS=1 } else { BIS=BIS}

BVS = BVS | BV;

Example:

```
FLOAT R2,R1;
```


FMADD

Description: The FMADD instruction multiplies one floating-point register value (RM) with a second floating-point register value (RN), adds the result to a third register(RD) and writes and places the result in register RD. The operation updates the floating-point arithmetic flags.

Syntax: FMADD <RD>, <RN>, <RM>;

<RD> Accumulation register for fused multiply add instruction

<RN> First operand register

<RM> Second operand register

Flags Updated:

BN Flag

BZ Flag

BV Flag

BIS Flag

BUS Flag

BVS Flag

Operation:

RD = RD + RN * RM

N = RD[31]

if (RD[30:0]==0) { BZ=1 } else { BZ=0}

if (UnbiasedExponent(RD) > 127) { BV=1 } else { BV=0}

if (UbiasedExponent(RD) < -126) { BUS=1 } else { BUS=BUS}

if (RM or RN == NAN) { BIS=1 } else { BIS=BIS}

BVS = BVS | BV;

Example:

```
FMADD R2,R1,R0
```

FMUL

Description: The FMUL instruction multiplies one floating-point register value (RM) with a second floating-point register value (RN) and places the result in register RD. The operation updates the floating-point arithmetic flags.

Syntax: FMUL <RD>, <RN>, <RM>;

<RD> Destination register

<RN> First operand register

<RM> Second operand register

Flags Updated: BN Flag

BZ Flag

BV Flag

BIS Flag

BUS Flag

BVS Flag

Operation:

```
RD=RN * RM
N = RD[31]
if (RD[30:0]==0) { BZ=1 } else { BZ=0}
if (UnbiasedExponent(RD) > 127) { BV=1 } else { BV=0}
if (UbiasedExponent(RD) < -126) { BUS=1 } else { BUS=BUS}
if (RM or RN == NAN) { BIS=1 } else { BIS=BIS}
BVS = BVS | BV;
```

Example:

```
FMUL R2,R1,R0;
```


FMSUB

Description: The FSUB instruction multiplies one floating-point register value (RM) with a second floating-point register value (RN), subtracts the result from a third register(RD) and writes and places the result in register RD. The operation updates the floating-point arithmetic flags.

Syntax: FMSUB <RD>, <RN>, <RM>

<RD> Accumulation register for fused multiply sub instruction

<RN> First operand register

<RM> Second operand register

Flags Updated: BN Flag

BZ Flag

BV Flag

BIS Flag

BUS Flag

BVS Flag

Operation:

RD = RD - RN * RM

N = RD[31]

if (RD[30:0]==0) { BZ=1 } else { BZ=0}

if (UnbiasedExponent(RD) > 127) { BV=1 } else { BV=0}

if (UbiasedExponent(RD) < -126) { BUS=1 } else { BUS=BUS}

if (RM or RN == NAN) { BIS=1 } else { BIS=BIS}

BVS = BVS | BV;

Example:

```
FMSUB R2,R1,R0;
```


FSUB

Description: The FSUB instruction subtracts one floating-point register value(RM) from another floating point register value(RN) and places the result in a third destination register(RD). The operation updates the floating-point arithmetic flags.

Syntax: FSUB <RD>, <RN>, <RM>

<RD> Destination register

<RN> First operand register

<RM> Second operand register

Flags Updated: BN Flag

BZ Flag

BV Flag

BIS Flag

BUS Flag

BVS Flag

Operation: RD=RN - RM

BN = RD[31]

if (RD[30:0]==0) { BZ=1 } else { BZ=0}

if (UnbiasedExponent(RD) > 127) { BV=1 } else { BV=0}

if (UbiasedExponent(RD) < -126) { BUS=1 } else { BUS=BUS}

if (RM or RN == NAN) { BIS=1 } else { BIS=BIS}

BVS = BVS | BV;

Example:

```
FSUB R2,R1,R0;
```


GID

Description: Disables all interrupts

Syntax: GID

Flags Updated: None

Operation: STATUS[1]=1

Example:
```
GID         ;
```


GIE

Description: Enables all interrupts in ILAT register, dependent on the per bit settings in the IMASK register.

Syntax: GIE

Flags Updated: None

Operation: STATUS[1]=0

Example:
```
GIE         ;
```


IADD

Description: The IADD instruction adds two 32-bit signed integer operands together and places the result in a third register. The operation updates the secondary status flags.

Syntax: IADD <RD>, <RN>, <RM>

<RD> Destination register

<RN> First operand register

<RM> Second operand register

Flags Updated: BN Flag

BZ Flag

Operation: RD=RN + RM

BN = RD[31]

if (RD[30:0]==0) { BZ=1 } else { BZ=0}

Example:

```
IADD R2, R2, R0;
```

IMADD

Description: The IMADD instruction multiplies one signed integer register value (RM) with a second signerd integer register value (RN), adds the result to a third register(RD) and writes and places the result in register RD. The operation updates the secondary arithmetic status flags.

Syntax: IMADD <RD>, <RN>, <RM>;

<RD> Accumulation register for fused multiply add instruction

<RN> First operand register

<RM> Second operand register

Flags Updated: BN Flag

BZ Flag

Operation: RD = RD + RN * RM

N = RD[31]

if (RD[30:0]==0) { BZ=1 } else { BZ=0}

Example:

```
IMADD R2, R1, R0
```

IMSUB

Description: The IMSUB instruction multiplies one signed integer register value (RM) with a second signed integer register value (RN), subtracts the result to a third register (RD) and writes and places the result in register RD. The operation updates the secondary arithmetic status flags.

Syntax: IMSUB <RD>, <RN>, <RM>

<RD>	Accumulation register for fused multiply sub instruction
<RN>	First operand register
<RM>	Second operand register

Flags Updated:

BN Flag

BZ Flag

Operation:

RD = RD - RN * RM

N = RD[31]

if (RD[30:0]==0) { BZ=1 } else { BZ=0}

Example:

IMSUB R2, R1, R0;

IMUL

Description: The IMUL instruction multiplies one signed integer register value (RM) with a second signed integer register value (RN) and places the result in register RD. The operation updates the secondary arithmetic status flags.

Syntax: IMUL <RD>, <RN>, <RM>;

<RD> Destination register

<RN> First operand register

<RM> Second operand register

Flags Updated: BN Flag

BZ Flag

Operation: RD=RN * RM

N = RD[31]

if (RD[30:0]==0) { BZ=1 } else { BZ=0}

Example:

```
IMUL R2, R1, R0;
```

ISUB

Description: The ISUB instruction subtracts one signed integer register value (RM) from another signed integer register value (RN) and places the result in a third destination register (RD). The operation updates the secondary arithmetic status flags.

Syntax: ISUB <RD>, <RN>, <RM>

<RD> Destination register

<RN> First operand register

<RM> Second operand register

Flags Updated: BN Flag

BZ Flag

Operation: RD=RN - RM

BN = RD[31]

if (RD[30:0]==0) { BZ=1 } else { BZ=0}

Example:

```
ISUB R2, R1, R0;
```


IDLE

Description: The instruction places the core in an idle state. The PC is halted and no more instructions are fetched until an interrupt wakes up the core.

Syntax: IDLE

Flags Updated: None

Operation:

```
STATUS[0]=0
while(!ILAT){
PC=PC;
}
```

Example:

```
IDLE ;
```


JALR

Description: The register-and-link jump instruction causes an unconditional jump to absolute address contained in Rn. Before jumping to the compute address, the next PC is saved in the link register (LR). The instruction allows for efficient support for subroutines and allows for jumping to any address supported by the instruction set architecture.

Syntax: JALR <RN>

<RN> Register with absolute address to jump to. registers

Flags Updated: None

Operation: LR = PC;
PC = RN;

Example:
```
MOV R0,#_labA  ;move label into register
JALR R0        ;save PC in LR and jump to labA
```


JR

Description: The register jump instruction causes an unconditional jump to the absolute address in register RN. This allows for jumping to any address supported by the instruction set architecture.

Syntax: JR <RN>;

<RN> Any one of the general-purpose registers.

Flags Updated: None

Flags Updated: None

Operation: PC = RN;

Example:

```
MOV R0,#_labA  ;move label into register
JR R0;         ;jump to _labA
```


LDR (DISPLACEMENT)

Description: The displacement mode LDR instruction loads a data from memory to a general-purpose register (RD). The memory address is the sum of the base register value (RN) and an immediate index offset. The base register is not modified by the load operation. The instruction supports loading of byte, short, word, and double data. Data must be aligned in memory according to the size of the data. For double data loads, only even RD registers can be used.

Syntax: LDR<size> <RD>, [<RN>, #+<IMM3>]

LDR<size> <RD>, [<RN>, #+/-<IMM11>]

<size> Byte(B), Half(H), Word(), or Double(D)

<RD> Destination register for the data loaded from memory.

<RN> Register containing the base address for the load instruction.

<IMM3 | IMM11> An unsigned 3 or 11 bit value shifted by 0, 1, 2 or 3 bits before being used to compute the address of the load store operation. The shifting amount depends on the size of the data being moved and allows for extending the range of the immediate value.

< - > The "–" option specified that the immediate value should be subtracted from the base address. This option is only available for the 11 bit immediate values instruction.

Flags Updated: None

Operation: address= RN +/- IMM << (log2(size_in_bits/8)) ;

RD=memory[address];

Example:

```
LDRB R31,[R2]  ;loads byte
LDR R0,[R2,#1]      ;loads word
```


LDR (INDEX)

Description: The index mode LDR loads data from memory to a general-purpose register (RD). The memory address is the sum of the base register (RN) and an index register (RM). The base register is not modified by the load operation. The instruction supports loading of byte, short, word, and double data. Data must be aligned in memory according to the size of the data. For double data loads, only even RD registers can be used.

Syntax: LDR<size> <RD>, [<RN>, +/-<RM>]

<size>	Byte(B), Half(H), Word(), or Double(D)
<RD>	Destination register for the word loaded from memory
<RN>	Register containing the base address for the load instruction
<RM>	Register containing the index address to add to the base address.
< - >	The "–" option specified that the index register should be subtracted from the base address.

Flags Updated: None

Operation: address= RN +/- RM;

RD=memory[address];

Example:

```
LDRB R31,[R2,R1]     ;loads byte
LDR R0,[R2,R1]       ;loads word
```


LDR (POSTMODIFY)

Description: The post-modify mode LDR loads data from memory to a general purpose register (RD). The memory address used is the value of the base register (RN). After loading the data from memory, the base value register (RN) is updated with the sum of the initial base value and the index value in (RM). The instruction supports loading of byte, short, word, and double data. Data must be aligned in memory according to the size of the data. For double data loads, only even RD registers can be used.

Syntax: LDR<size> <RD>, [<RN>], +/-<RM>

<size>	Byte(B), Half(H), Word(), or Double(D)
<RD>	Destination register for the word loaded from memory.
<RN>	Register containing the base address for the load instruction.
<RM>	Register containing the index address for the load instruction.
< - >	The "–" option specified that the index register should be subtracted from the base address.

Flags Updated: None

Operation:

address= RN;

RD=memory[address];

RN=RN +/- RM;

Example:

```
LDRS R31,[R2],R1     ;loads short, updates R2
LDRD R0,[R2],R1      ;loads double, updates R2
```


LDR (DISPLACEMENT-POSTMODIFY)

Description: The post-modify mode LDR allows a word to be loaded from memory to a general-purpose register (RD). The memory address used is the value of the base register (RN). After loading the data from memory, the base value register (RN) is updated with the sum/subtraction of the initial base value and the immediate index value (IMM11). The instruction supports loading of byte, short, word, and double data. Data must be aligned in memory according to the size of the data. For double data loads, only even RD registers can be used.

Syntax: LDR<size> <RD>, [<RN>], #+/-<IMM11>

<size>	Byte(B), Half(H), Word(), or Double(D)
<RD>	Destination register for the data loaded from memory
<RN>	Register containing the base address for the load instruction
<IMM11>	An unsigned 11 bit value shifted by 0, 1, 2 or 3 bits before being used to compute the address of the load store operation. The shifting allows for extending the range of the immediate value. The value is added to the value of <RN> to form the address in memory from which the word is loaded.
< - >	The "–" option specified that the index address should be subtracted from the base address.

Flags Updated: None

Operation:

address= RN;

RD=memory[address];

RN=RN +/- IMM11 << (log2(size_in_bits/8));

Example:

```
LDRS R31,[R2],#1     ;loads short, updates R2
LDRD R0,[R2],#4      ;loads double, updates R2
```


LSL

Description: The LSL instruction performs a logical shift left of the RN operand based on the shift value (OP2). OP2 is a 5 bit unsigned immediate value or an unsigned shift value contained within the first 5 bits of operand register RM. Zeros fill the bit positions vacated by the shifted RN word. The result is placed in register RD.

Syntax: LSL <RD>, <RN>, <RM>

LSL <RD>, <RN>, #IMM5

<RD>	Destination register
<RN>	First operand register
<RM>	Second operand register
<IMM5>	Five bit unsigned immediate value

Flags Updated: AN

AZ

AV

AC

Operation: RD = RN << <OP2>

AN = RD[31]

AV = 0

AC = 0

if (RD[31:0]==0) { AZ=1 } else { AZ=0}

Example:

```
LSL  R0,R1,R2  ;
LSL  R0,R1,#3  ;
```


LSR

Description: The LSR instruction performs a logical shift right of the RN operand based on the shift value (OP2). OP2 is a 5 bit unsigned immediate value or a unsigned shift value contained within the first 5 bits of operand register RM. Zeros fill the bit positions vacated by the shifted RN word. The result is placed in register RD.

Syntax: LSR <RD>, <RN>, <RM>

LSR <RD>, <RN>, #IMM5

<RD> Destination register

<RN> First operand register

<RM> Second operand register

<IMM5> Five bit unsigned immediate value

Flags Updated: AN

AZ

AV

AC

Operation: RD = RN >> <OP2>

AN = RD[31]

AV = 0

AC = 0

if (RD[31:0]==0) { AZ=1 } else { AZ=0}

Example:

```
LSR  R0,R1,R2  ;
LSR  R0,R1,#3  ;
```


MBKPT (LABS)

Description: The MBKPT instruction sends a halt signal to all cores in the system. It allows all cores to stop at approximately the same time, simplifying multicore code debugging easier. The propagation of the MBKPT signal across the chip can be blocked by setting the appropriate edge bits in the MESHCONFIG register.

Syntax: MBKPT

Flags Updated: None

Operation: Halts all cores within the group (sets DEBUGSTATUS[0] to “1”)

MOV<COND>

Description The MOV instruction conditionally copies the contents of the source register (RN) into the destination register (RD). The condition codes are the same as those of the conditional branch instructions. A MOV without any condition field moves register RN to register RD regardless of the state of the flags.

Syntax: MOV<cond> <RD>, <RN>

<cond> One of the 15 condition codes. Legal condition codes include:

EQ, NE, GT, GTE, LTE, LT, LTU, LTEU,GTU, GTEU, BEQ, BNE, BLT, BLTE. If no argument is specified, the copy always happens.

<RD> Destination register

<RN> Source register for move operation.

Flags Updated: None

Operation: IF (Passed) <COND>)) then

RD = RN

Example:

```
MOVEQ     R2,R0 ;copies R0 to R2 if the EQ
MOV       R3,R1 ;copies R1 to R3
```


MOV (IMMEDIATE)

Description: The MOV immediate instruction copies an unsigned immediate constant in the destination register (RD).

Syntax: MOV <RD>, #<IMM8>;

MOV <RD>, #<IMM16>;

<RD> Destination register for move operation.

<IMM8> An 8-Bit unsigned immediate value.

<IMM16> A 16-Bit unsigned immediate value.

Flags Updated: None

Operation: RD=<imm>

Example:

```
MOV R0,#25          ;Sets R0 to 25
```


MOVT (IMMEDIATE)

Description: The MOVT immediate instruction copies an unsigned immediate constant in the destination register (RD).

Syntax: MOVT <RD>, #<IMM16>;

<RD> Destination register for move operation.

<IMM16> A 16-Bit unsigned immediate value.

Flags Updated: None

Operation: RD=Rd(low) | (<imm16> << 16)

Example:

```
MOV R0,%low(0x90000000)       ;sets all 32 bits
MOVT R0,%high(0x90000000)     ;sets upper 16-bits
```

MOVFS

Description: The MOVFS instruction copies a value from a special core control register to a general-purpose register.

Syntax: MOVFS <RD>, <SPECIAL>;

<SPECIAL> Special Register to copy value from

<RD> General-purpose destination register for move operation

Flags Updated: None

Operation: RD = SPECIAL

Example:

```
MOVFS R0,CONFIG     ;copies CONFIG value to R0
```


MOVTS

Description: The MOVTS instruction copies a value from a general purpose register file to a core control registers.

Syntax: MOVTS <SPECIAL>, <RN>

<SPECIAL> Special Register to copy value into

<RN> General-purpose source register for move operation

Flags Updated: None

Operation: SPECIAL = RN

Example:

```
MOVTS CONFIG,R0     ;copies R0 to CONFIG register
```


NOP

Description: The instruction does nothing, but holds an instruction slot.

Syntax: NOP

Flags Updated: None

Operation: None

Example:
```
NOP         ;
```


ORR

Description: The ORR instruction logically ors the operand in register RN with the operand in register RM and places the result in register RD.

Syntax: ORR <RD>, <RN>, <RM>

<RD> Destination register

<RN> First operand register

<RM> Second operand register

Flags Updated: AN Flag

AZ Flag

AV Flag

AC Flag

Operation: RD = RN | RM

AN = RD[31]

AV = 0

AC = 0

if (RD[31:0]==0) { AZ=1 } else { AZ=0}

Example:

```
ORR R2,R1,R0    ;
```


RTI

Description: The RTI instruction causes the address in the IRET register to be restored to the PC register, a clearing of the corresponding bit in the IPEND register. All actions are carried out as a single atomic operation.

Syntax: RTI;

Flags Updated: None

Operation: IPEND[i]=0; where i is the current interrupt level being serviced

STATUS[1]=0;

PC=IRET;

<execute instruction at PC>

Example:

```
RTI  ;
```


RTS (alias instruction)

Description: This is an alias instruction for JR <LR>. When branching to a subroutine using the BL or JALR instruction, the next instruction PC is saved in register R14 (LR). It is used to return from a subroutine/function in the program.

Syntax: RTS;

Flags Updated: None

Operation: PC=R14

Example:

```
RTS  ;
```


SUB

Description: The SUB instruction subtracts an integer register value (OP2) from an integer value in register (RN). The OP2 operand can be an immediate value (SIMM3 | SIMM11) or register value (RM).

Syntax: SUB <RD>, <RN>, <RM>

SUB <RD>, <RN>, #SIMM3

SUB <RD>, <RN>, #SIMM11

<RD>	Destination register
<RN>	First operand register
<RM>	Second operand register
<SIMM3 | SIM11>	Three or eleven bit signed immediate value.

Flags Updated: AN Flag

AZ Flag

AV Flag

AC Flag

Operation:

```
RD = RN - <OP2>
AN = RD[31]
AC = BORROW
if (RD[31:0]==0) { AZ=1 } else { AZ=0}
if ((RD[31] & ~RM[31] & RN[31]) | (RD[31] & ~RM[31] & RN[31]) )
        { OV=1 }
else { OV=0 }
AVS = AVS | AV
```

Example:

```
SUB R2,R1,R0    ;
```


STR (DISPLACEMENT)

Description: The displacement mode STR stores a word to memory from a general purpose register (RD). The memory address is the sum of the base register value (RN) and an immediate index offset. The base register is not modified by the store operation. The instruction supports storing of byte, short, word, and double data. Data must be aligned in memory according to the size of the data. For double data stores, only even RD registers can be used.

Syntax: STR<size> <RD>, [<RN>, #+<IMM3>]

STR<size> <RD>, [<RN>, #+/-<IMM11>]

<size>	Byte(B), Half(H), Word(), or Double(D)
<RD>	Source register for the word store to memory.
<RN>	Register containing the base address for the store instruction.
<IMM3 | IMM11>	An unsigned 3 or 11 bit value shifted by 0, 1, 2 or 3 bits before being used to compute the address of the load store operation. The shifting allows for extending the range of the immediate value. The value is added to the value of <RN> to form the address in memory to which the word is stored.
< - >	The "–" option specified that the index register should be subtracted from the base address.

Flags Updated: None

Operation: address= RN +/- IMM << (log2(size_in_bits/8));

memory[address]=RD;

Example:

```
STRB R31,[R2,#1]     ;stores byte to addr in R2
STR  R0,[R2,#0x4]    ;stores word to addr in R2
```


STR (INDEX)

Description: The index mode STR stores a word to memory from a general-purpose register (RD). The memory address is the sum of a base register (RN) and an index register.(RM) The base register is not modified by the store operation. The instruction supports loading of byte, short, word, and double data. Data must be aligned in memory according to the size of the data. For double data loads, only even RD registers can be used.

Syntax: STR<size> <RD>, [<RN>, +/-<RM>]

<size>	Byte(B), Half(H), Word(), or Double(D)\
<RD>	Source register for the word stored to memory.
<RN>	Register containing the base address for the store instruction.
<RM>	Register containing the index address for the store instruction.
< - >	The "–" option specified that the index register should be subtracted from the base address.

Flags Updated: None

Operation: address= RN +/- RM;

memory[address]=RD;

Example:

```
STRB R31,[R2,R1]     ;stores byte to addr in R2
STR  R0,[R2,R1]      ;stores word to addr in R2
```


STR (POSTMODIFY)

Description: The postmodify STR instruction stores a word in memory from a general purpose register (RD). The memory address used is the value of the base register (RN). After storing the the data in memory, the base value register (RN) is updated with the sum of the initial base value(RN) and the index value in (RM). The instruction supports loading of byte, short, word, and double data. Data must be aligned in memory according to the size of the data. For double data loads, only even RD registers can be used.

Syntax: STR<size> <RD>, [<RN>], +/-<RM>

<size>	Byte(B), Half(H), Word(), or Double(D)
<RD>	Source register for the word stored to memory.
<RN>	Register containing the base address for the store instruction.
<RM>	Register containing the index address for the store instruction.
< - >	The "–" option specified that the index register should be subtracted from the base address.

Flags Updated: None

Operation:

address= RN;

memory[address]=RD;

RN=RN +/- RM;

Example:

```
STRS R31,[R2], R1   ;stores short to addr in R2
STRD R0,[R2], R3    ;stores double to addr in R2
```


STR (DISPLACEMENT-POSTMODIFY)

Description: The postmodify STR instruction stores a word in memory from a general purpose register (RD). The memory address used is the value of the base register (RN). After storing the the data in memory, the base value register (RN) is updated with the sum of the initial base value(RN) and the index value in (IMM11). The instruction supports loading of byte, short, word, and double data. Data must be aligned in memory according to the size of the data. For double data loads, only even RD registers can be used.

Syntax: STR<size> <RD>, [<RN>], +/-<IMM11>

<size>	Byte(B), Half(H), Word(), or Double(D)
<RD>	Source register for the word stored to memory.
<RN>	Register containing the base address for the store instruction.
<IMM11>	An unsigned 11 bit value shifted by 0, 1, 2 or 3 bits before being used to compute the address of the load store operation. The shifting allows for extending the range of the immediate value. The value is added to the value of <RN> to form the address in memory to which the word is stored.
< - >	The "–" option specified that the index register should be subtracted from the base address.

Flags Updated: None

Operation:

address= RN;

memory[address]=RD;

RN=RN +/- IMM11 << (log2(size_in_bits/8));

Example:

```
STRS R31,[R2],#2     ;stores short to addr in R2
STRD R0,[R2],#1      ;stores double to addr in R2
```


SYNC (LABS)

Description: The SYNC instruction sends an interrupt pulse to all cores on the chip. It allows for a secondary method of syncing up all cores to start operating at approximately the same time. The propagation of the SYNC signal can be blocked by setting the appropriate edge bits in the MESHCFG register.

Syntax: SYNC

Flags Updated: None

Operation: Sets the ILAT[0] of all cores within a work group to “1”.

TRAP

Description: The TRAP instruction causes the processor to halt and wait for external inputs. The immediate field within the instruction opcode is not processed by the hardware but can be used by software such as a debugger or operating system to find out the reason for the TRAP instruction.

Syntax: TRAP <IMM5>

<IMM5> An unsigned 5 bit value. The following list indicates the different codes that can be used with the TRAP instruction to indicate what action to take by the operating system, debugger, or other software infrastructure.

0-2 = reserved
3 = program exit indicator
4 = indicates success, can be used to indicate "test passed"
5 = indicates assertion, test "failed"
6 = reserved
7 = initiates system call

In the case of TRAP 7, a system call is initiated. In this case, a sub argument needs to be passed in R3 indicating what further action to take, based on the following table. Arguments to the system calls are passed in Register R0-R2.

Function	R0	R1	R2	R3
File Open	Path Name Pointer	0	0	2
File Close	File Descriptor	0	0	3
Read	File Descriptor	Buffer Pointer	Buffer Length	4
Write	File Descriptor	Buffer Pointer	Buffer Length	5
File Lseek	File Descriptor	File Offset	Whence	6
File Unlink	Path Name Pointer	0	0	7
Fstat	Path Name Pointer	Status Pointer	0	10
Stat	File Descriptor	Status Pointer	0	15

Flags Updated: None

Operation: Halts processor;

Example:
```
TRAP 0    ;Halt processor to prepare for write
```


TESTSET

Description: The TESTSET instruction does an atomic "test-if-not-zero", then conditionally writes on any memory location within the Epiphany architecture. The absolute address used for the test and set instruction must be located within the on-chip local memory and must be greater than 0x00100000. The instruction tests the value of a specific memory location and if that value is zero, writes in a new value from the local register file. If the value at the memory location was already set to a non-zero value, then the value is returned to the register file, but the memory location is left unmodified.

Syntax: TESTSET RD, [RN, +/-RM];

Flags Updated: None

Operation:

```
if ([RN+/-RM]) {
  RD= ([RN+/-RM])
}
else{
 ([RN+/-RM])=RD
 RD=0;
}
```

Flags Updated: None

Example:

```
/*example of trying to lock on value in memory*/
_loop: MOV R2, R3           ; value to write
       TESTSET R2, [R0, R1]; test-set
       SUB R2, R2, #0       ; check result
       BNE_loop             ; keep trying
```


WAND (LABS)

Description: The WAND instruction sets flag bit [3] in the STATUS register and moves on to the next instruction. When all cores in a group have set their respective wand bits, then an interrupt is generated on the WAND-flag interrupt line. The instruction can be used to create distributed multicore barriers.

Syntax: WAND

Flags Updated: None

Operation: STATUS[3]=1

Example:

```
WAND;
IDLE; /*wait for every core in the group to execute WAND*/
…
/*in ISR, clear STATUS[3] bit*/
```


Appendix B: Register Set Reference

Register Summary

This appendix contains detailed descriptions for all the registers within the Epiphany core architecture. Complete 32 bit addresses are constructed by combining the local 20 bit addresses shown in the tables with the MSB aligned 12 bit core ID.

Table 27: eCore Registers

Local Address	Register Name	Access	Comment
0xF0000→ 0xF00FC	R0-R63	RD/WR	General purpose registers
0xF0400	CONFIG	RD/WR	Core configuration
0xF0404	STATUS	RD/WR	Core status
0xF0408	PC	RD/WR	Program counter
0xF040C	DEBUGSTATUS	RD	Debug status
0xF0414	LC	RD/WR	Hardware loop counter
0xF0418	LS	RD/WR	Hardware loop start address
0xF041C	LE	RD/WR	Hardware loop end address
0xF0420	IRET	RD/WR	Interrupt PC return value
0xF0424	IMASK	RD/WR	Interrupt mask
0xF0428	ILAT	RD/WR	Interrupt latch
0xF042C	ILATST	WR	Alias for setting interrupts
0xF0430	ILATCL	WR	Alias for clearing interrupts
0xF0434	IPEND	RD/WR	Interrupts currently in process
0xF0440	FSTATUS	WR	Alias for writing to all STATUS bits

0xF0448	DEBUGCMD	WR	Debug command register
0xF070C	RESETCORE	WR	Per core software reset

Table 28: Event Timer Registers

Address	Register Name	Access	Comment
0xF0438	CTIMER0	RD/WR	Core timer0
0xF043C	CTIMER1	RD/WR	Core timer1

Table 29: Processor Control Registers

Address	Register Name	Access	Comment
0xF0604	MEMSTATUS	RD/WR	Memory protection status
0xF0608	MEMPROTECT	RD/WR	Memory protection configuration

Table 30: DMA Registers

Address	Register Name	Access	Address
0xF0500	DMA0CONFIG	RD/WR	DMA channel0 configuration
0xF0504	DMA0STRIDE	RD/WR	DMA channel0 stride
0xF0508	DMA0COUNT	RD/WR	DMA channel0 count
0xF050C	DMA0SRCADDR	RD/WR	DMA channel0 source address
0xF0510	DMA0DSTADDR	RD/WR	DMA channel0 destination address
0xF0514	DMA0AUTO0	RD/WR	DMA channel0 slave lower data
0xF0518	DMA0AUTO1	RD	DMA channel0 slave upper data

0xF051C	DMA0STATUS	RD/WR	DMA channel0 status
0xF0520	DMA1CONFIG	RD/WR	DMA channel1 configuration
0xF0524	DMA1STRIDE	RD/WR	DMA channel1 stride
0xF0528	DMA1COUNT	RD/WR	DMA channel1 count
0xF052C	DMA1SRCADDR	RD/WR	DMA channel1 source address
0xF0530	DMA1DSTADDR	RD/WR	DMA channel1 destination address
0xF0534	DMA1AUTO0	RD/WR	DMA channel1 slave lower data
0xF0538	DMA1AUTO1	RD/WR	DMA channel1 slave upper data
0xF053C	DMA1STATUS	RD/WR	DMA channel1 status

Table 31: Mesh Node Control Registers

Address	Register Name	Access	Comment
0xF0700	MESHCONFIG	RD/WR	Mesh node configuration
0xF0704	COREID	RD	Processor node ID
0xF0708	MULTICAST	RD/WR	Multicast configuration
0xF0710	CMESHROUTE	RD/WR	cMesh routing configuration
0xF0714	XMESHROUTE	RD/WR	xMesh routing configuration
0xF0718	RMESHROUTE	RD/WR	rMesh routing configuration

Table 32: CMESHROUTE Register

CMESHROUTE: 0xF0710		
Bit	Name	Function
[2:0]	NORTH_CONFIG	0xx: normal routing 100: block northbound transactions 101: send northbound transactions east 110: send northbound transactions south 111: send northbound transactions west
[5:3]	EAST_CONFIG	0xx: normal routing 100: block northbound transactions 101: send northbound transactions south 110: send northbound transactions west 111: send northbound transactions north
[8:6]	SOUTH_CONFIG	0xx: normal routing 100: block northbound transactions 101: send northbound transactions west 110: send northbound transactions north 111: send northbound transactions east
[11:9]	WEST_CONFIG	0xx: normal routing 100: block northbound transactions 101: send northbound transactions north 110: send northbound transactions east 111: send northbound transactions south

COREID

The row-column coordinate of the processor nodes is contained in the read-only COREID register, which is accessible by software using the "MOVFS RN, COREID" instruction. The COREID register facilitates writing code that is independent of processor nodes and that can be easily mapped to any node within the Epiphany architecture.

Table 33: COREID Register

COREID: 0xF0704		
Bits	Name	Function
[5:0]	COLUMN_ID	Core column ID
[11:6]	ROW_ID	Core row ID
[31:12]	RESERVED	N/A

CONFIG

Table 34: CONFIG Register

CONFIG: 0xF0400		
Bit	Name	Function
[0]	RMODE	IEEE Floating-Point Truncate Rounding Mode 0 = Round to nearest even rounding 1 = Truncate rounding
[1]	IEN	Invalid floating-point exception enable 0 = Exception turned off 1 = Exception turned on
[2]	OEN	Overflow floating-point exception enable 0 = Exception turned off 1 = Exception turned on
[3]	UEN	Underflow floating-point exception enable 0 = Exception turned off 1 = Exception turned on
[7:4]	CTIMER0CFG	Controls the events counted by CTIMER0. 0000 = off 0001 = clk 0010 = idle cycles 0011 = reserved 0100 = IALU valid instructions

		0101 = FPU valid instructions 0110 = dual issue clock cycles 0111 = load (E1) stalls 1000 = register dependency (RA) stalls 1001 = reserved 1010 = local memory fetch stalls 1011 = local memory load stalls 1100 = external fetch stalls 1101 = external load stalls 1110 = mesh traffic monitor 0 1111 = mesh traffic monitor 1
[11:8]	CTIMER1CFG	Timer1 mode, same description as for CTIMER0. A 0011 configuration selects the carry out from TIMER0, effectively creating a 64 bit timer (NOTE: not available in Epiphany-III).
[15:12]	CTRLMODE	This register controls certain routing modes within the eMesh. More information can be found in eMesh chapter. 0000:Normal routing mode 0100: DMA channel0 last transaction indicator 1000: DMA channel1 last transaction indicator 1100: Message mode routing **(LABS)** 0001: Force routing to the NORTH at destination 0101: Force routing to the EAST at destination 1001: Force routing to the SOUTH at destination 1101: Force routing to the WEST at destination

		xx10: Reserved 0011: Multicast routing **(LABS)** 1011: Reserved 0111: Reserved 1111: Reserved
[16]	RESERVED	N/A
[19:17]	ARITHMODE	Selects the operating mode of the data path unit.(“FPU”) 000 = 32bit IEEE float point mode 100 = 32bit signed integer mode All other modes reserved.
[21:20]	RESERVED	N/A
[22]	LPMODE	0=Only minimal clock gating in idle mode 1=Aggressive power down in idle mode (Recommended).
[25-23]	RESERVED	N/A
[26]	TIMERWRAP	0=Timer stops when it reaches 0x0 1=Timer resets to 0xFFFFFFFF when it reaches 0x0 and keeps going. (only available in Epiphany-IV) **(LABS)**
[31:27]	RESERVED	

CTIMER0

Table 35: CTIMER0 Register

CTIMER0: 0xF0438		
Bits	Name	Function
[31:0]	CTIMER0	Complete 32-bit timer or lower 32-bits of 64-bit timer

CTIMER1

Table 36: CTIMER1 Register

CTIMER1: 0xF043C		
Bits	Name	Function
[31:0]	CTIMER1	Complete 32-bit timer or upper 32-bits of 64-bit timer

DMAxAUTO0 (LABS)

The DMA AUTO registers are used in DMA slave mode. In slave mode, the rate of transactions is controlled by the rate of writes being to the AUTODMA register rather than by the DMA transaction master. In the case of double-word data transfers, DMAxAUTO0 receives the lower 32 bits and DMAxAUTO1 receives the upper 32 bits.

Table 37: DMAxAUTO0 Register

DMA0AUTO0: 0xF0514 DMA1AUTO0: 0xF0534		
Bits	Name	Function
[31:0]	DMAxAUTO0	Lower 32-bits of the DMA slave mode receiver register pair

DMAxAUTO1 (LABS)

Table 38: DMAxAUTO0 Register

DMA0AUTO1: 0xF0518 DMA1AUTO1: 0xF0538		
Bits	Name	Function
[31:0]	DMAxAUTO0	Lower 32-bits of the DMA slave mode receiver register pair

DMAxCONFIG

The DMA configuration register is used to configure the type of DMA transfer. The following table shows the configuration options for each channel in the DMA engine.

Table 39: DMACONFIG Register

DMA0CONFIG: 0xF0500 DMA1CONFIG: 0xF0520		
Bit	Name	Function
[0]	DMAEN	Turns on DMA channel. 1=enabled 0=disabled
[1]	MASTER	Sets up DMA channel to work in master made 1=master mode 0=slave mode
[2]	CHAINMODE	Sets up DMA in chaining mode so that a new descriptor is

		automatically fetched from the next descriptor address at the end of the current configuration. 1=Chain mode 0=One-shot mode
[3]	STARTUP	Used to kick start the DMA configuration. When this bit is set to 1, the DMA sequencer looks at bits [31:16] to find the descriptor address to fetch the complete DMA configuration from. Once the descriptor has been completely fetched, the DMA will start data transfers. 1=Fetch descriptor 0=Normal operation
[4]	IRQEN	Enables interrupt at the end of the complete DMA channel. In the case of chained interrupts, the interrupt is set before the next descriptor is fetched. 1=Enable interrupt at end of DMA transfer 0=Disable interrupt at end of DMA transfer.
[6:5]	DATASIZE	Size of data transfer. 00=byte, 01=half-word, 10=word, 11=double-word
[9:7]	RESERVED	N/A
[10]	MSGMODE	Attach a special message to the last data item of a DMA channel transfer. If the destination address is local memory, then the transition to DMA_IDLE only occurs after the last data item has returned. If the destination address is in another core, then a message interrupt (IRQ5) is sent along

		with the last data item. **(LABS)**
[11]	RESERVED	N/A
[12]	SHIFT_SRC_IN	Left shift inner loop source stride address by 16 bits **(LABS)**
[13]	SHIFT_DST_IN	Left shift inner loop destination stride address by 16 bits **(LABS)**
[14]	SHIFT_SRC_OUT	Left shift outer loop source stride address by 16 bits **(LABS)**
[15]	SHIFT_DST_OUT	Left shift outer loop destination stride address by 16 bits **(LABS)**
[31:16]	NEXT_PTR	Address of next DMA descriptor for normal operation. Address of immediate descriptor to fetch in case of startup mode.

DMAxCOUNT

This register is used to set up the number of transactions in the inner and outer loops of the DMA transaction. The upper 16 bits specify the outer loop of the DMA transfer and the and lower 16 bits of the register specify the number of inner loops. The outer and inner loops must be set to a value of one or greater. The DMA block transfer is complete when the DMACOUNT register reaches zero. The inner count value is cleared to the initial count every time the outer loop is decremented.

Table 40: DMACOUNT Register

DMA0COUNT: 0xF0508 DMA1COUNT: 0xF0528		
Bits	Name	Function

[15:0]	INNER_COUNT	Transactions remaining within inner loop.
[31:16]	OUTER_COUNT	Number of outer loop iterations remaining. (“2D”)

DMAxDSTADDR

This register contains the 32-bit address of the transaction currently being transferred. The address can be a local address (bits [31:20] all zero) or a global address. The register gets loaded when the descriptor is fetched from memory and is updated at the completion of every transaction. The updated address is equal to the old destination address added with the value in the destination field in the stride register.

Table 41: DMADSTADDR Register

DMA0DSTADDR: 0xF0510 DMA1DSTADDR: 0xF0530		
Bits	Name	Function
[31:0]	DSTADDR	Current transaction destination address to write to

DMAxSRCADDR

This register contains the 32-bit source address of the transaction currently being transferred. The address can be a local address (bits [31:20] all zero) or a global address. The register gets loaded when the descriptor is fetched from memory and is updated at the completion of every transaction. The updated address is equal to the old source address added with the value in the destination field in the stride register.

Table 42: DMASRCADDR Register

DMA0SRCADDR: 0xF050C DMA1SRCADDR: 0xF052C		
Bits	Name	Function

[31:0]	SRC_ADDR	Current transaction source address to read from.

DMAxSTATUS

Table 43: DMASTATUS Register

DMA0STATUS: 0xF051C DMA1STATUS: 0xF053C		
Bit	Name	Function
[3:0]	DMASTATE	0x0=DMA idle 0x5=DMA active 0x6=DMA in slave mode waiting for transaction 0xA=Waiting for last data item to return to local memory. Only relevant for MSGMODE. 0xB=DMA in transfer pause state. 0xD=DMA configuration error. Occurs when the DMA configuration register is written while the DMA is not in in an IDLE state. All other DMA states are temporary in nature and are not meaningful to the user.
[15:4]	RESERVED	N/A
[31:16]	CURR_PTR	The address of DMA descriptor currently being processed.

DMAxSTRIDE

The register contains two signed 16-bit values specifying the stride, in bytes, used to update the source and destination address register after a completed transaction. The lower 16 bits specify source address register update stride and the upper 16 bits specify the destination address stride. At the end of an inner-loop turn, this register is loaded with the outer-loop stride values to make address adjustments of the source and destination addresses before continuing with the next inner

loop of data transfer. Before the next inner loop starts, the stride register is reloaded with the inner-loop stride values. The stride values are specified in bytes and should match the type of transfers being done. All DMA transactions must be aligned appropriately in memory.

Table 44: DMASTRIDERegister

DMA0STRIDE: 0xF0504 DMA1STRIDE: 0xF0524		
Bits	Name	Function
[15:0]	SRC_STRIDE	Value to add to the DMAxSRCADDR after each transaction.
[31:16]	DST_STRIDE	Value to add to the DMAxDSTADDR after each transaction.

DEBUGCMD

A write only alias register used to place control the debug state of the Epiphany core from an external agent.

Table 45: DEBUGCMD Register

DEBUGCMD: 0xF0448		
Bits	Name	Function
[1:0]	COMMAND	00: Force the processor into a “running” state (i.e. resume) 01 : Force the processor into a “halt” state (i.e. halt)

DEBUGSTATUS

Table 46: DEBUGSTATUS Register

DEBUGSTATUS: 0xF040C		
Bits	Name	Function
[0]	HALT	0: Processor operating normally 1: Processor in “halt” state

[1]	EXT_PEND	0: No external load or fetch pending 1: External load or fetch pending
[2]	MBKPT_FLAG	0: No multicore breakpoint active 1: Multicore breakpoint active **(LABS)**
[31:3]	RESERVED	N/A

FSTATUS (LABS)

Table 47: FSTATUS Status Register

FSTATUS: 0xF0440		
Bits	Name	Function
[31:0]	STATUS	Write to all bits in STATUS register. Normally in writing to the STATUS register, bits [2:0] are not writable.

ILAT

The ILAT register records all interrupt events. All events are positive edge-triggered, meaning that there is no need to hold the interrupt bit high until the event has been completed. Each bit in the ILAT register (except for the user interrupt at bit 9) is tied to a specific hardware event. The ILAT register can be accessed directly or through the address aliases ILATST and ILATCL.

Table 48: ILAT Register

ILAT: 0xF0428		
Bits	Name	Function
[9:0]	ILAT	Latched interrupts waiting to enter CPU
[31:10]	RESERVED	N/A

ILATST

An alias for the ILAT register that allows bits within the ILAT register to be set individually. Writing a “1” to an individual bit of the ILATST register will set the corresponding ILAT bit to “1”. Writing a “0” to an individual bit will have no effect on the ILAT register. The ILATST alias cannot be read.

Table 49: ILATST Register Alias

ILATST: 0xF042C		
Bits	Name	Function
[9:0]	ILATST	Safely sets individual bits of the ILAT register.
[31:10]	RESERVED	N/A

ILATCL

An alias for the ILAT register that allows bits within the ILAT register to be cleared individually. Writing a “1” to an individual bit of the ILATCL register will clear the corresponding ILAT bit to “0”. Writing a “0” to an individual bit will have no effect on the ILAT register. The ILATST alias cannot be read.

Table 50: ILATCL Register Alias

ILATCL: 0xF0430		
Bits	Name	Function
[9:0]	ILATCL	Safely clears individual bits of the ILAT register.
[31:10]	RESERVED	N/A

IMASK

This is a masking register for blocking interrupts on a per-interrupt basis. All interrupts are latched by the ILAT register but can be blocked from reaching the program sequencer by setting the appropriate bit in the IMASK register. At each bit position, a “1” means the interrupt is masked.

Table 51: IMASK Register

IMASK: 0xF0424		
Bits	Name	Function
[9:0]	ILAT	Latched interrupts waiting to enter CPU
[31:10]	RESERVED	N/A

IRET

When an interrupt is serviced, the program counter of the upcoming sequential instruction is saved in the IRET register. The value in the IRET register is used by the RTI instruction to return to the original thread at a later time. For nested interrupt service routines, the IRET should be saved on the stack.

Table 52: IRET Register

IRET: 0xF0420		
Bits	Name	Function
[31:0]	IRET	The saved program counter (PC) at the time of the interrupt
[31:10]	RESERVED	N/A

IPEND

This is a status register that keeps track of the interrupt service routines currently being processed. A bit is set when the interrupt enters the core and redirects the program flow and is cleared by the software executing an RTI instruction. The lowest numbered bit set to "1" indicates the currently serviced interrupt. Only interrupts in the ILAT register with a number less than the lowest bit in the IPEND register reach the program sequencer. This register can be used to implement nested interrupts. The register should never be directly written by a program.

Table 53: IPEND Register

IPEND: 0xF0434

Bits	Name	Function
[9:0]	IPEND	Maintains record of all interrupts currently being serviced.
[31:10]	RESERVED	N/A

LC (LABS)

Table 54: LC Register

LC: 0xF0414		
Bits	Name	Function
[31:0]	LOOP_COUNT	Current loop count, decremented when LE==PC.

LE (LABS)

Table 55: LE Register

LE: 0xF041C		
Bits	Name	Function
[31:0]	LOOP_END	Loop end address

LS (LABS)

Table 56: LS Register

LS: 0xF0418		
Bits	Name	Function
[31:0]	LOOP_START	Loop start address

MEMPROTECT (LABS)

NOTE: Only bits [7:0] available in Epiphany III.

Table 57: MEMPROTECT Register

MEMPROTECT: 0xF0608		
Bits	Name	Function
[0]	PAGE0	1: Addr 0x0000 → 0x0FFF set as read-only memory
[1]	PAGE1	1: Addr 0x1000 → 0x1FFF set as read-only memory
[2]	PAGE2	1: Addr 0x2000 → 0x2FFF set as read-only memory
[3]	PAGE3	1: Addr 0x3000 → 0x3FFF set as read-only memory
[4]	PAGE4	1: Addr 0x4000 → 0x4FFF set as read-only memory
[5]	PAGE5	1: Addr 0x5000 → 0x5FFF set as read-only memory
[6]	PAGE6	1: Addr 0x6000 → 0x6FFF set as read-only memory
[7]	PAGE7	1: Addr 0x7000 → 0x7FFF set as read-only memory
[9:8]	RESERVED	N/A
[10]	DIS_EXT_RD	1: Disable reading core address range from external source
[11]	DIS_EXT_WR_MMR	1: Disable writing to MMR register from external source
[12]	DIS_EXT_WR_MEM	1: Disable writing to local memory from external source
[13]	DIS_CORE_CWR	1: Disables core's ability to write to other on-chip core
[14]	DIS_CORE_XWR	1: Disables core's ability to write off-chip
[15]	EXC_EN	1: Enables interrupt from MEMPROTECT exception.

MEMSTATUS (LABS)

NOTE: Only bit [2] available in Epiphany III.

Table 58: MEMSTATUS Register

MEMSTATUS: 0xF0604		
Bits	Name	Function
[1:0]	RESERVED	N/A
[2]	MEM_FAULT	Memory protection fault from one of 8 local pages
[9:3]	RESERVED	N/A
[10]	READ_BREACH	Read from external agent attempted with DIS_EXT_RD==1
[11]	WRITE_BREACH	Read from external agent attempted with DIS_EXT_WR==1
[12]	CWRITE_BREACH	Write to on-chip cores attempted with DIS_CORE_CWR=1
[13]	XWRITE_BREACH	Write to on-chip cores attempted with DIS_CORE_XWR=1

Table 59: Mesh Configuration Register

MESHCONFIG: 0xF0700		
Bit	Name	Function
[0]	RESERVED	N/A
[1]	LPMODE	0=Minimal clock gating in idle mode (high power) 1=Aggressive power down in idle mode (recommended)
[2]	RESERVED	N/A
[3]	RESERVED	N/A
[7:4]	MESHEVENT1	Configures mesh node input events to track on cMesh. The even monitored can be programmed as an input to CTIMER0 or CTIMER1. 0000 = off 0001 = reserved 0010 = any wait 0011 = core wait 0100 = south wait 0101 = north wait 0110 = west wait 0111 = east wait 1000 = southeast wait 1001 = northwest wait 1010 = south access 1011 = north access

		1100 = west access 1101 = east access 1110 = core access 1111 = any access (available in Epiphany-IV only)
[11:8]	MESHEVENT0	Same configuration format as MESHEVENT1
[12]	WESTEDGE	Blocks sync, wand, and mbkpt from propagating west
[13]	EASTEDGE	Blocks sync, wand, and mbkpt from propagating east
[14]	NORTHEDGE	Blocks sync, wand, and mbkpt from propagating north
[15]	SOUTHEDGE	Blocks sync, wand, and mbkpt from propagating south

MULTICAST (LABS)

Table 60: MULTICAST Register

MULTICAST: 0xF0704		
Bits	Name	Function
[11:0]	MULTICAST_ID	ID to match to destination address[31:20] in the case of an incoming multicast write transaction
[31:12]	RESERVED	N/A

PC

Table 61: PC Register

COREID: 0xF0408		
Bits	Name	Function
[31:0]	PROGRAM_COUNTER	Contains next sequential PC to be executed

RMESHROUTE (G4-LABS)

RMESHROUTE: 0xF0718		
Bit	Name	Function
[2:0]	NORTH_CONFIG	0xx: normal routing 100: block northbound transactions 101: send northbound transactions east 110: send northbound transactions south 111: send northbound transactions west
[5:3]	EAST_CONFIG	0xx: normal routing 100: block northbound transactions 101: send northbound transactions south 110: send northbound transactions west 111: send northbound transactions north
[8:6]	SOUTH_CONFIG	0xx: normal routing 100: block northbound transactions 101: send northbound transactions west 110: send northbound transactions north 111: send northbound transactions east
[11:9]	WEST_CONFIG	0xx: normal routing 100: block northbound transactions 101: send northbound transactions north 110: send northbound transactions east

111: send northbound transactions south

RESETCORE (LABS)

A write-only register alias used to bring the core in and out of a reset state.

Table 62: RESETCORE Register

RESETCORE: 0xF070C		
Bits	Name	Function
[0]	RESET	0: Core is taken out of reset 1: Core is put into a reset state

STATUS

The STATUS register contains information regarding the execution status of the CPU.

Table 63: STATUS Register

STATUS: 0xF0404			
Bit	Flag Name	Updated By	Function
[0]	ACTIVE	Interrupt, IDLE	Core active indicator 0=core idle, 1=core active
[1]	GID	RTI, Interrupt, GIE, GID	Global interrupts disabled indicator 0 = all interrupts enabled 1 = all interrupts disabled
[2]	RESERVED	N/A	N/A
[3]	WAND	WAND instruction	Multicore communication **(LABS)**
[4]	AZ	Integer Instructions	Integer Zero Flag

[5]	AN	Integer Instructions	Integer Negative Flag
[6]	AC	Integer Instructions	Integer Carry Flag
[7]	AV	Integer Instructions	Integer Overflow Flag
[8]	BZ	Floating-Point Instructions	Floating-Point Zero Flag
[9]	BN	Floating-Point Instructions	Floating-Point Negative Flag
[10]	BV	Floating-Point Instructions	Alternate Overflow Flag
[11]	RESERVED	N/A	N/A
[12]	AVS	Integer Instructions	Sticky Integer Overflow
[13]	BIS	Floating-Point Instructions	Sticky Floating-Point Invalid
[14]	BVS	Floating-Point Instructions	Sticky Floating-Point Overflow
[15]	BUS	Floating-Point Instructions	Sticky Floating-Point Underflow
[19:16]	EXCAUSE	Instructions and events	Software exception cause
[31:0]	RESERVED	N/A	N/A

The register reflects the state of the processor and should always be saved on entering an interrupt service routine. The STATUS register can be written using the MOVTS instruction or by directly writing to the register from an external host. The sticky flags can only be cleared through the MOVTS instruction or an externally generated write transaction. Status bits [2:0] are ready only bits controlled by the operational state of the CPU, but can be written forcibly through the FSTATUS alias.

XMESHROUTE (G4-LABS)

Table 64: XMESHROUTE Register

XMESHROUTE: 0xF0714		
Bit	Name	Function
[2:0]	NORTH_CONFIG	0xx: normal routing 100: block northbound transactions 101: send northbound transactions east 110: send northbound transactions south 111: send northbound transactions west
[5:3]	EAST_CONFIG	0xx: normal routing 100: block northbound transactions 101: send northbound transactions south 110: send northbound transactions west 111: send northbound transactions north
[8:6]	SOUTH_CONFIG	0xx: normal routing 100: block northbound transactions 101: send northbound transactions west 110: send northbound transactions north 111: send northbound transactions east
[11:9]	WEST_CONFIG	0xx: normal routing 100: block northbound transactions 101: send northbound transactions north 110: send northbound transactions east 111: send northbound transactions south

Appendix C: Instruction Set Decode

This chapter specifies the Epiphany instruction set operation codes (opcodes).

Table 65: Opcode Field Summary

Field	Function
I23:0	Immediate value for branch Instructions
C3:0	Condition codes for branch instructions
RN, RM, RD	Register operands
SUB	Specifies that second operand is subtracted from first operand in load/store instructions. (1=subtract, 0=add)
S	Specifies store (1=store, 0=load)
B1,B0	Load/store operation size 00=Byte mode 10=Halfword mode 10=Word mode 11=Double mode
S4:S0	Immediate field used by shift instructions
M1,M0	MMR register group 00= Core Registers (0xF04XXX) 01= DMA Registers (0xF05XXX) 10=Memory Protection Registers (0xF06XXX) 11=Processor Node Configuration Register (0xF07XXX)
T4:T0	See TRAP instruction for description

Table 66: Epiphany Instruction Decode Table

	[31:28]				[27:24]				[23:20]				[19:16]				[15:12]				[11:8]				[7:4]				[3:0]			
B<COND> (16)																	I7	I6	I5	I4	I3	I2	I1	I0	C3	C2	C1	C0	0	0	0	0
B<COND> (32)	I23	I22	I21	I20	I19	I18	I17	I16	I15	I14	I13	I12	I11	I10	I9	I8	I7	I6	I5	I4	I3	I2	I1	I0	C3	C2	C1	C0	1	0	0	0
LDR/STR (DISP) (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	I2	I1	I0	B1	B0	S	0	1	0	0
LDR/STR (INDEX) (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	B1	B0	S	0	0	0	1
LDR/STR (PM) (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	B1	B0	S	0	1	0	1
LDR/STR (DISP) (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	0	SUB	I10	I9	I8	I7	I6	I5	I4	I3	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	I2	I1	I0	B1	B0	S	1	1	0	0
LDR/STR (PM-DISP) (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	1	SUB	I10	I9	I8	I7	I6	I5	I4	I3	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	I2	I1	I0	B1	B0	S	1	1	0	0
LDR/STR (INDEX) (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3	0	0	SUB					Rm2	Rm1	Rm0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	B1	B0	S	1	0	0	1
TESTSET (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3	0	1	SUB					Rm2	Rm1	Rm0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	B1	B0	0	1	0	0	1
LDR/STR (PM) (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3	0	0	SUB					Rm2	Rm1	Rm0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	B1	B0	S	1	1	0	1
MOV (IMM) (16)																	Rd2	Rd1	Rd0	I7	I6	I5	I4	I3	I2	I1	I0	0	0	0	1	1
MOV (IMM) (32)	Rd5	Rd4	Rd3	0	I15	I14	I13	I12	I11	I10	I9	I8					Rd2	Rd1	Rd0	I7	I6	I5	I4	I3	I2	I1	I0	0	1	0	1	1
MOVT (IMM) (32)	Rd5	Rd4	Rd3	1	I15	I14	I13	I12	I11	I10	I9	I8					Rd2	Rd1	Rd0	I7	I6	I5	I4	I3	I2	I1	I0	0	1	0	1	1
ADD(IMM) (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	I2	I1	I0	0	0	1	0	0	1	1
SUB (IMM) (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	I2	I1	I0	0	1	1	0	0	1	1
ADD(IMM) (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3			I10	I9	I8	I7	I6	I5	I4	I3	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	I2	I1	I0	0	0	1	1	0	1	1
SUB (IMM) (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3			I10	I9	I8	I7	I6	I5	I4	I3	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	I2	I1	I0	0	1	1	1	0	1	1
LSR (IMM) (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	S4	S3	S2	S1	S0	0	0	1	1	0
LSL(IMM)(16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	S4	S3	S2	S1	S0	1	0	1	1	0
ASR(IMM)(16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	S4	S3	S2	S1	S0	0	1	1	1	0
BITR(16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	S4	S3	S2	S1	S0	1	1	1	1	0
LSR (IMM) (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3							0	1	1	0	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	S4	S3	S2	S1	S0	0	1	1	1	1
LSL(IMM)(32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3							0	1	1	0	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	S4	S3	S2	S1	S0	1	1	1	1	1
ASR(IMM)(32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3							1	1	1	0	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	S4	S3	S2	S1	S0	0	1	1	1	1
BITR(32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3							1	1	1	0	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	S4	S3	S2	S1	S0	1	1	1	1	1
ADD (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	0	1	1	0	1	0
SUB (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	1	1	1	0	1	0
AND (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	1	0	1	1	0	1	0
ORR (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	1	1	1	1	0	1	0
EOR (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	0	0	1	0	1	0
ASR (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	1	1	0	1	0	1	0
LSR (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	1	0	0	1	0	1	0
LSL (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	1	0	1	0	1	0
ADD (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				1	0	1	0	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	0	1	1	1	1	1
SUB (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				1	0	1	0	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	1	1	1	1	1	1
AND (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				1	0	1	0	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	1	0	1	1	1	1	1
ORR (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				1	0	1	0	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	1	1	1	1	1	1	1
EOR (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				1	0	1	0	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	0	0	1	1	1	1
ASR (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				1	0	1	0	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	1	1	0	1	1	1	1
LSR (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				1	0	1	0	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	1	0	0	1	1	1	1
LSL (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				1	0	1	0	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	1	0	1	1	1	1
FADD/IADD (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	0	0	0	1	1	1
FSUB/ISUB (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	0	1	0	1	1	1
FMUL/IMUL (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	1	0	0	1	1	1
FMADD/IMADD (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	1	1	0	1	1	1
FMSUB/IMSUB (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	1	0	0	0	1	1	1
FLOAT (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	0	0	0	1	0	1	0	1	1	1
FIX (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	0	0	0	1	1	0	0	1	1	1
FABS (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	0	0	0	1	1	1	0	1	1	1
FADD/IADD (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				0	1	1	1	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	0	0	1	1	1	1
FSUB/ISUB (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				0	1	1	1	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	0	1	1	1	1	1
FMUL/IMUL (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				0	1	1	1	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	1	0	1	1	1	1
FMADD/IMADD (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				0	1	1	1	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	0	1	1	1	1	1	1
FMSUB/IMSUB (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				0	1	1	1	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	Rm2	Rm1	Rm0	1	0	0	1	1	1	1
FLOAT (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				0	1	1	1	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	0	0	0	1	0	1	1	1	1	1
FIX (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				0	1	1	1	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	0	0	0	1	1	0	1	1	1	1
FABS (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3	Rm5	Rm4	Rm3				0	1	1	1	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	0	0	0	1	1	1	1	1	1	1
MOV<COND> (16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	0	0	C3	C2	C1	C0	0	0	1	0
MOV<COND> (32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3							0	0	1	0	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	0	0	C3	C2	C1	C0	1	1	1	1
MOVTS(16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	0	1	0	0	0	0	0	0	1	0
MOVFS(16)																	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	0	1	0	0	0	1	0	0	1	0
MOVTS(32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3					M1	M0	0	0	1	0	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	0	1	0	0	0	0	1	1	1	1
MOVFS(32)	Rd5	Rd4	Rd3	Rn5	Rn4	Rn3					M1	M0	0	0	1	0	Rd2	Rd1	Rd0	Rn2	Rn1	Rn0	0	1	0	0	0	1	1	1	1	1
JR (16)																				Rn2	Rn1	Rn0	0	1	0	1	0	0	0	0	1	0
JALR (16)																				Rn2	Rn1	Rn0	0	1	0	1	0	1	0	0	1	0
JR (32)				Rn5	Rn4	Rn3							0	0	1	0				Rn2	Rn1	Rn0	0	1	0	1	0	0	1	1	1	1
JALR (32)				Rn5	Rn4	Rn3							0	0	1	0				Rn2	Rn1	Rn0	0	1	0	1	0	1	1	1	1	1
GIE (16)																							0	1	1	0	0	1	0	0	1	0
GID (16)																							1	1	1	0	0	1	0	0	1	0
NOP (16)																							0	1	1	0	1	0	0	0	1	0
IDLE (16)																							0	1	1	0	1	1	0	0	1	0
BKPT (16)																							0	1	1	1	0	0	0	0	1	0
MBKPT (16)																							1	1	1	1	0	0	0	0	1	0
SYNC (16)																							0	1	1	1	1	1	0	0	1	0
RTI (16)																							0	1	1	1	0	1	0	0	1	0
WAND (16)																							0	1	1	0	0	0	0	0	1	0
TRAP (16)																	T5	T4	T3	T2	T1	T0	1	1	1	1	1	0	0	0	1	0
UNIMPL (16)													1	1	1	1							0	0	0	0	0	0	1	1	1	1

Appendix D: Architecture Evolutionary Changes

This chapter documents the differences between the Epiphany-III and Epiphany-IV versions of the Epiphany architecture.

Table 67: Epiphany Architecture Changes

Feature	Register	Epiphany-III	Epiphany-IV
Software Exception	STATUS[19:16]	0100=unimplemented 0001=swi 0010=unaligned 0101=illegal access 0011=fpu exception	1111=unimplemented 1110=swi 1101=unaligned 1100=illegal access 0111=fpu excpetion
Timer	CONFIG[11:8]	N/A	0011=Enables 64 bit counter
Mesh events	MESHCONFIG[7:4]	1111=N/A	1111=Counts any access
Routing Configuration	CMESHROUTE	N/A	Available
Routing Configuration	RDMESHROUTE	N/A	Available
Routing Configuration	XMESHROUTE	N/A	Available

Appendix E: Architecture Manual Changes

Table 68: Reference Change Log

Revision	Changes
3.11.09.09	Initial document release
3.13.9.29	Disclosed MSGMODE for DMA Disclosed shifted stride for DMA Disclosed routing CTRLMODE field in CONFIG register Disclosed hardware loop chapter Disclosed debug functionality chapter Disclosed WAND, MBKPT, SYNC features Disclosed multicast feature Disclosed Epiphany-IV routing configuration registers Fixed lots of typos (an probably added some more..) Moved all register descriptions to central alphabetical chapter Hyperlinked register description table
14.02.21	Syntax fixes, formatting Fixed SDK diagram Changed naming version scheme
14.03.11	Clarified which features are “LABS” features

Appendix F: Copyright Information

Adapteva Inc.
1666 Massachusetts Ave, Suite 14
Lexington, MA 02420
USA