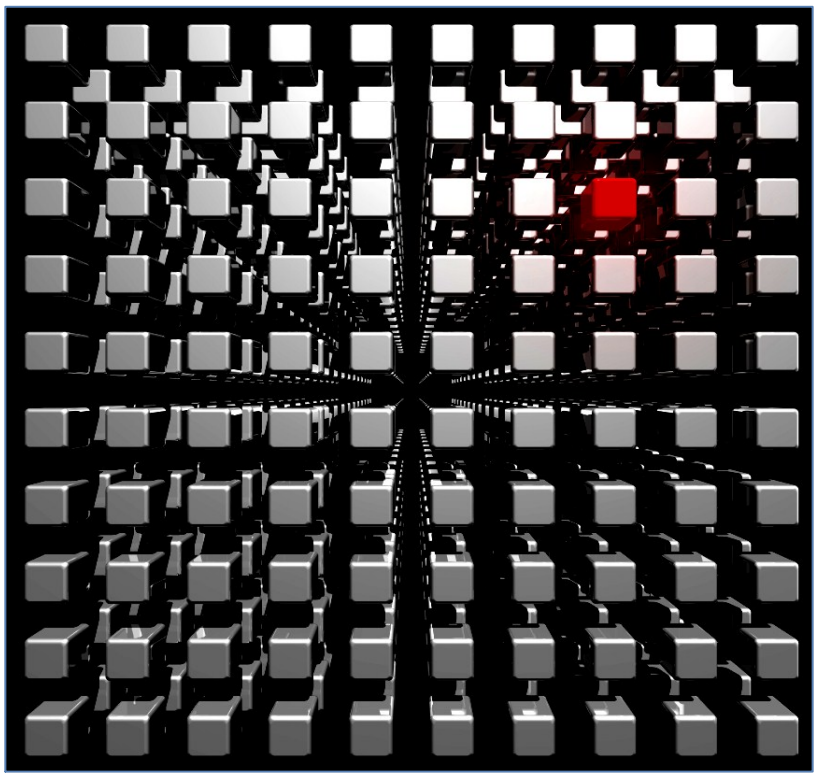
# Epiphany™ Architecture Reference

# (G3)



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

## Table of Contents


**1 Introduction ..... 10**

**2 Programming Model ..... 13**

2.1 Programming Model Introduction ..... 13
2.2 Parallel Programming Example ..... 14

**3 Software Development Environment ..... 16**

**4 Memory Architecture ..... 17**

4.1 Memory Address Map ..... 17
4.2 Memory Order Model ..... 19
4.3 Endianness ..... 20
4.4 Load/Store Alignment Restrictions ..... 21
4.5 Program-Fetch Alignment Restrictions ..... 21

**5 eMesh™ Network-On-Chip ..... 22**

5.1 Network Topology ..... 22
5.2 Routing Protocol ..... 24
5.3 Read Transactions ..... 25
5.4 Direct Inter-Core Communication ..... 26
5.5 Arbitration Scheme ..... 27
5.6 Data Sizes and Alignment ..... 27

**6 Processor Node Subsystem ..... 28**

6.1 Processor Node Overview ..... 28
6.1.1 eCore™ CPU ..... 29
6.1.2 Local Memory ..... 29
6.1.3 Direct Memory Access (DMA) Engine ..... 29
6.1.4 Event Timers ..... 29
6.1.5 Network Interface ..... 29
6.2 Mesh-Node Crossbar Switch ..... 30
6.3 Mesh-Node Arbitration ..... 32
6.4 Mesh-Node ID (COREID) ..... 32
6.5 Memory Protection Register (MEMPROTECT) ..... 33

**7 eCore™ CPU ..... 34**

7.1 Overview ..... 34
7.1.1 Program Sequencer ..... 34
7.1.2 Register File ..... 35
7.1.3 Integer ALU ..... 35
7.1.4 Floating-Point Unit ..... 36
7.1.5 Interrupt Controller ..... 36
7.1.6 Debug Unit ..... 36

7.2 Data Types ..... 37
7.2.1 Signed Integer Representation ..... 37
7.2.2 Unsigned Integer Representation ..... 37
7.2.3 Floating-Point Data Types ..... 37
7.3 Local Memory Map ..... 40
7.4 General Purpose Registers ..... 40
7.5 eCore Configuration Register ..... 43
7.6 eCore Status Register ..... 44
7.7 The Epiphany Instruction Set ..... 48
7.7.1 Branch Instructions ..... 48
7.7.2 Load/Store Instructions ..... 49
7.7.3 Integer Instructions ..... 50
7.7.4 Floating-Point Instructions ..... 50
7.7.5 Secondary Signed Integer Instructions ..... 50
7.7.6 Register Move Instructions ..... 51
7.7.7 Program Flow Instructions ..... 51
7.7.8 Instructions Set Summary ..... 51
7.8 Interrupt and Exception Handling ..... 57
7.8.1 Interrupt Routine Link (IRET) ..... 59
7.8.2 Interrupt Mask (IMASK) ..... 59
7.8.3 Interrupt Latch (ILAT) ..... 59
7.8.4 Interrupt Latch Set Alias (ILATST) ..... 59
7.8.5 Interrupt Latch Clear Alias (ILATCL) ..... 59
7.8.6 Interrupt Service Pending Status (IPEND) ..... 59
7.8.7 Status Register (STATUS) ..... 60
7.8.8 Global Enabling/Disabling of Interrupts ..... 60
7.8.9 Software Interrupts ..... 60
7.9 Pipeline Description ..... 62
7.10 Dual-Issue Scheduling Rules ..... 63
7.11 Branch Penalties ..... 67
**8 Direct Memory Access (DMA) ..... 68**
8.1 Overview ..... 68
8.2 Configuration Register (DMA{0,1}CONFIG) ..... 69
8.3 Count Register (DMA{0,1}COUNT) ..... 70
8.4 Stride Register (DMA{0,1}STRIDE) ..... 70
8.5 Source Address Register (DMA{0,1}SRCADDR) ..... 71
8.6 Destination Address Register (DMA{0,1}DSTADDR) ..... 71
8.7 AUTODMA Register (DMA{0,1}AUTO0 / DMA{0,1}AUTO1) ..... 71
8.8 Status Register (DMA{0,1}STATUS) ..... 71
8.9 DMA Descriptors ..... 72
8.10 DMA Channel Arbitration ..... 72

8.11 DMA Usage Restrictions ..... 72
8.12 DMA Transfer Example ..... 73
**9 Event Timers ..... 74**
9.1 CTIMER0 ..... 74
9.2 CTIMER1 ..... 74
**Appendix A: Instruction Set Reference ..... 75**
ADD ..... 76
AND ..... 77
ASR ..... 78
B<COND> ..... 79
BL ..... 80
BITR ..... 81
BKPT ..... 82
EOR ..... 83
FABS ..... 84
FADD ..... 85
FIX ..... 86
FLOAT ..... 87
FMADD ..... 88
FMUL ..... 89
FMSUB ..... 90
FSUB ..... 91
GID ..... 92
GIE ..... 93
IADD ..... 94
IMADD ..... 95
IMSUB ..... 96
IMUL ..... 97
ISUB ..... 98
IDLE ..... 99
JALR ..... 100
JR ..... 101
LDR (DISPLACEMENT) ..... 102
LDR (INDEX) ..... 103
LDR (POSTMODIFY) ..... 104
LDR (DISPLACEMENT-POSTMODIFY) ..... 105
LSL ..... 106
LSR ..... 107
MOV<COND> ..... 108
MOV (IMMEDIATE) ..... 109

MOVT (IMMEDIATE) ..... 110
MOVFS ..... 111
MOVTS ..... 112
NOP ..... 113
ORR ..... 114
RTI ..... 115
RTS (alias instruction) ..... 116
SUB ..... 117
STR (DISPLACEMENT) ..... 118
STR (INDEX) ..... 119
STR (POSTMODIFY) ..... 120
STR (DISPLACEMENT-POSTMODIFY) ..... 121
TRAP ..... 122
TESTSET ..... 123
**Appendix B: Register Description Reference ..... 124**

# List of Figures

FIGURE 1: AN IMPLEMENTATION OF THE EPIPHANY ARCHITECTURE ....10
FIGURE 2: EMESH™ NETWORK-ON-CHIP OVERVIEW ....11
FIGURE 3: MATRIX MULTIPLICATION DATA FLOW ....15
FIGURE 4: EPIPHANY SOFTWARE DEVELOPMENT KIT....16
FIGURE 5: EPIPHANY GLOBAL ADDRESS MAP ....17
FIGURE 6: EPIPHANY SHARED MEMORY MAP ....18
FIGURE 7: EMESH™ NETWORK TOPOLOGY....23
FIGURE 8: EMESH™ ROUTING EXAMPLE ....26
FIGURE 9: POINTER MANIPULATION EXAMPLE ....27
FIGURE 10: PROCESSOR NODE OVERVIEW ....28
FIGURE 11: PROGRAM MEMORY LAYOUT OPTIMIZED FOR SIZE ....31
FIGURE 12: PROGRAM MEMORY LAYOUT OPTIMIZED FOR SPEED....31
FIGURE 13: ECORE CPU OVERVIEW....34
FIGURE 14: INTERRUPT SERVICE ROUTINE OPERATION....57
FIGURE 15: INTERRUPT LATENCY DEPENDENCY ON EXTERNAL READ....61
FIGURE 16: PIPELINE GRAPHICAL VIEW....63

# List of Tables


TABLE 1: MEMORY TRANSACTION ORDERING RULE ....20
TABLE 2: LOAD AND STORE MEMORY-ALIGNMENT RESTRICTIONS ....21
TABLE 3: ROUTING PROTOCOL SUMMARY ....25
TABLE 4: PROCESSOR NODE ACCESS PRIORITIES ....32
TABLE 5: PROCESSOR NODE ID FORMAT ....32
TABLE 6: MEMORY PROTECTION REGISTER ....33
TABLE 7: IEEE SINGLE-PRECISION FLOATING-POINT DATA TYPES ....38
TABLE 8: ECORE LOCAL MEMORY MAP SUMMARY ....40
TABLE 9: GENERAL-PURPOSE REGISTERS ....41
TABLE 10: CORE CONFIGURATION REGISTER ....43
TABLE 11: CORE STATUS REGISTER ....44
TABLE 12: CONDITION CODES ....49
TABLE 13: INSTRUCTION SET SYNTAX ....52
TABLE 14: BRANCHING INSTRUCTIONS ....53
TABLE 16: INTEGER INSTRUCTIONS ....54
TABLE 17: FLOATING-POINT INSTRUCTIONS ....55
TABLE 18: REGISTER MOVE INSTRUCTIONS ....56
TABLE 19: PROGRAM FLOW INSTRUCTIONS ....56
TABLE 20: INTERRUPT SUPPORT SUMMARY ....58
TABLE 21: PIPELINE STAGE DESCRIPTION ....62
TABLE 22: PARALLEL SCHEDULING RULES ....64
TABLE 23: IALU INSTRUCTION SEQUENCES ....65
TABLE 24: FPU INSTRUCTION SEQUENCES ....65
TABLE 25: LOAD INSTRUCTION SEQUENCES ....66
TABLE 26: STALLS INDEPENDENT OF INSTRUCTION SEQUENCE ....66
TABLE 27: BRANCH PENALTIES ....67
TABLE 28: DMA TRANSFER TYPES ....68
TABLE 29: DMA CONFIGURATION REGISTER ....69
TABLE 30: DMA STATUS REGISTER ....71
TABLE 31: DMA DESCRIPTORS ....72
TABLE 32: ECORE REGISTERS ....124
TABLE 33: DMA REGISTERS ....125
TABLE 34: EVENT TIMER REGISTERS ....126
TABLE 35: PROCESSOR CONTROL REGISTERS ....126

# Preface

This document describes the third generation (G3) of 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

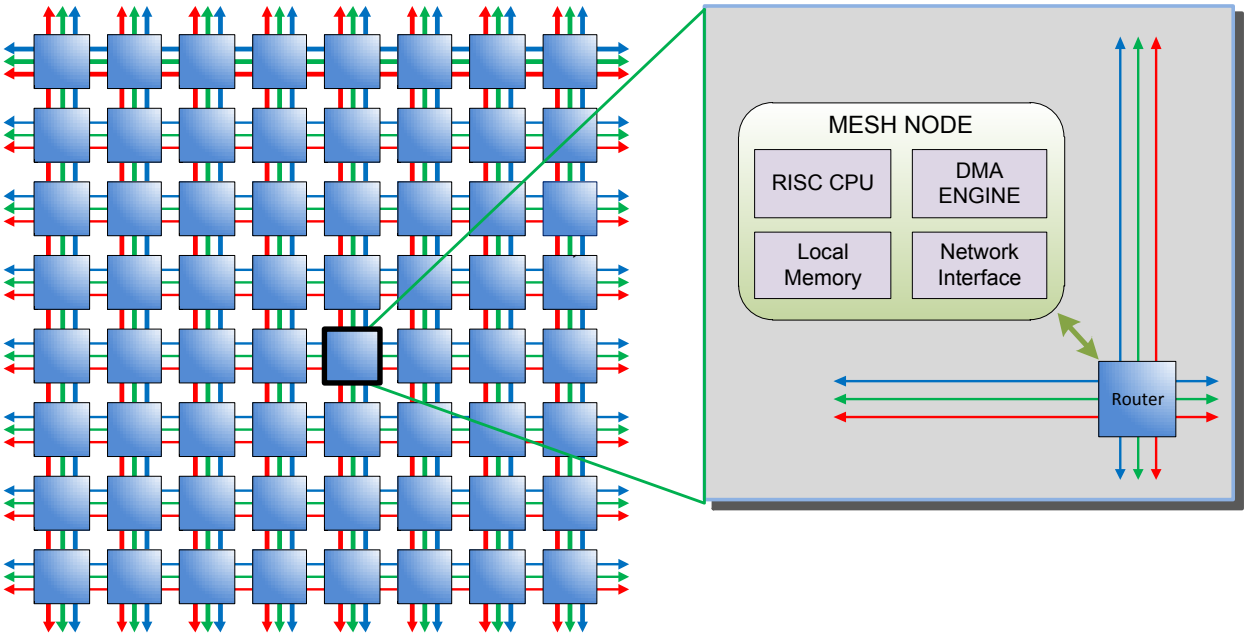
- Epiphany SDK Reference (epiphany_sdk_reference.pdf): The development tools and run-time library for the Epiphany architecture, available from Adapteva.
- Epiphany Quick-Start Guide (epiphany_quickstart.pdf): A tutorial on how to get up and running quickly with the Epiphany development tools, available from Adapteva.

# 1 Introduction

The Epiphany™ architecture defines a multicore, scalable, shared-memory, parallel computing fabric. It consists of a 2D array of mesh 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 cock cycle.
- Local memory in each mesh node that supports 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.

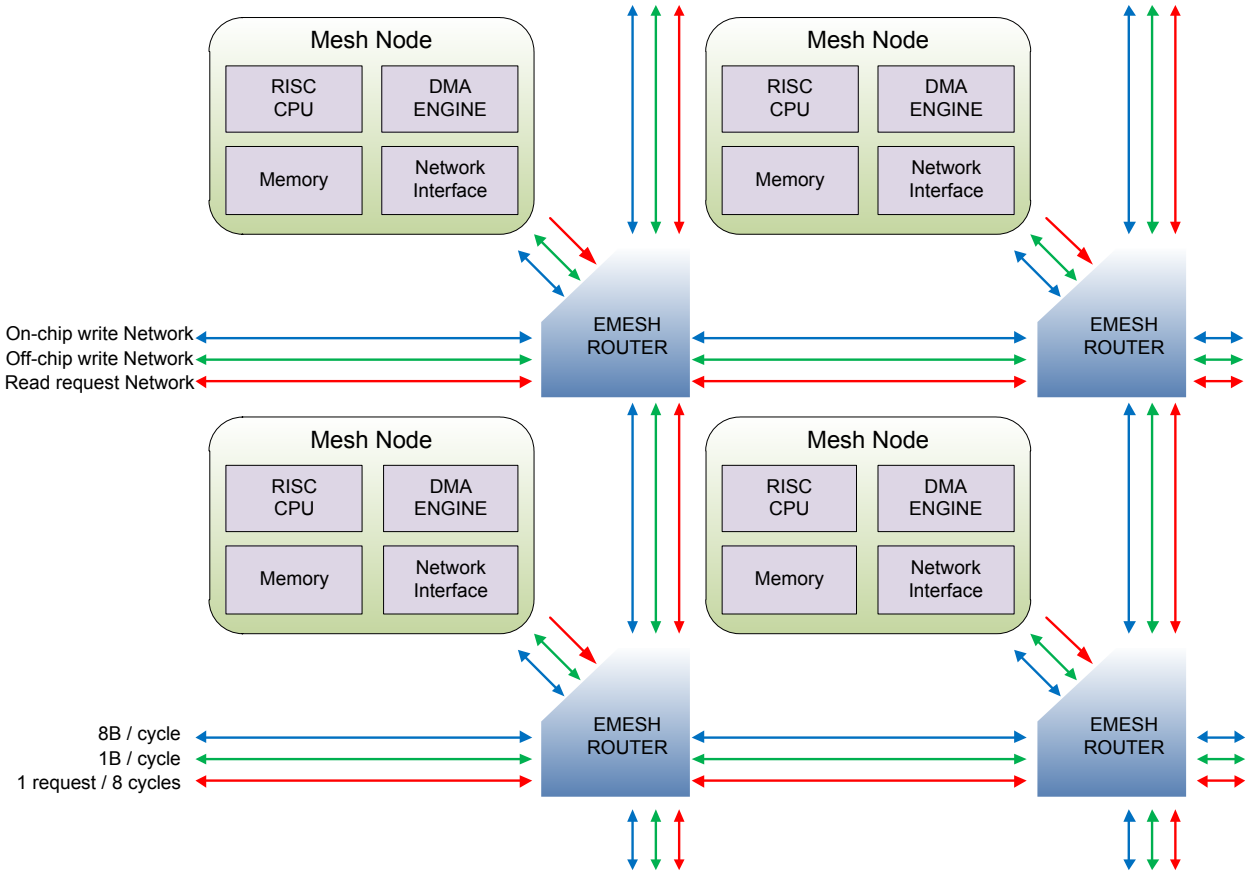
**Figure 1: An Implementation of the Epiphany Architecture**




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:

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

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];
```

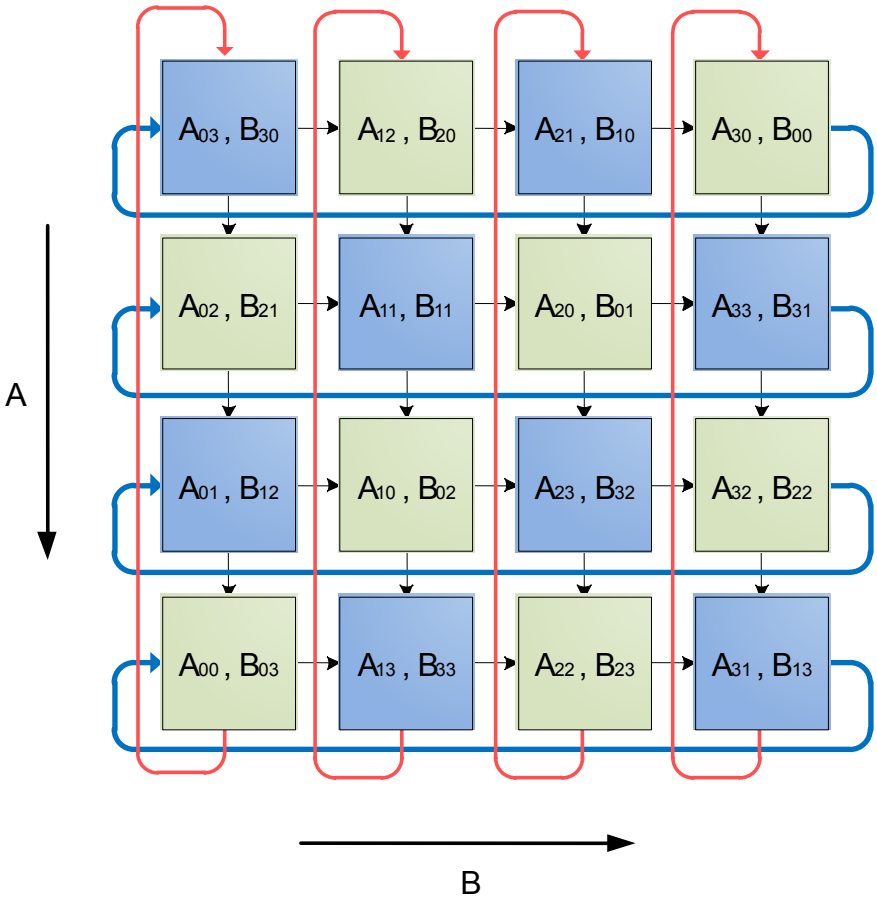
The code above can be written in standard C/C++ and compiled to run on a single core, with matrices A, B, and C placed in the core's local memory. In this simple programming example, there is no difference between the Epiphany architecture and any other single threaded processor platform.

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.

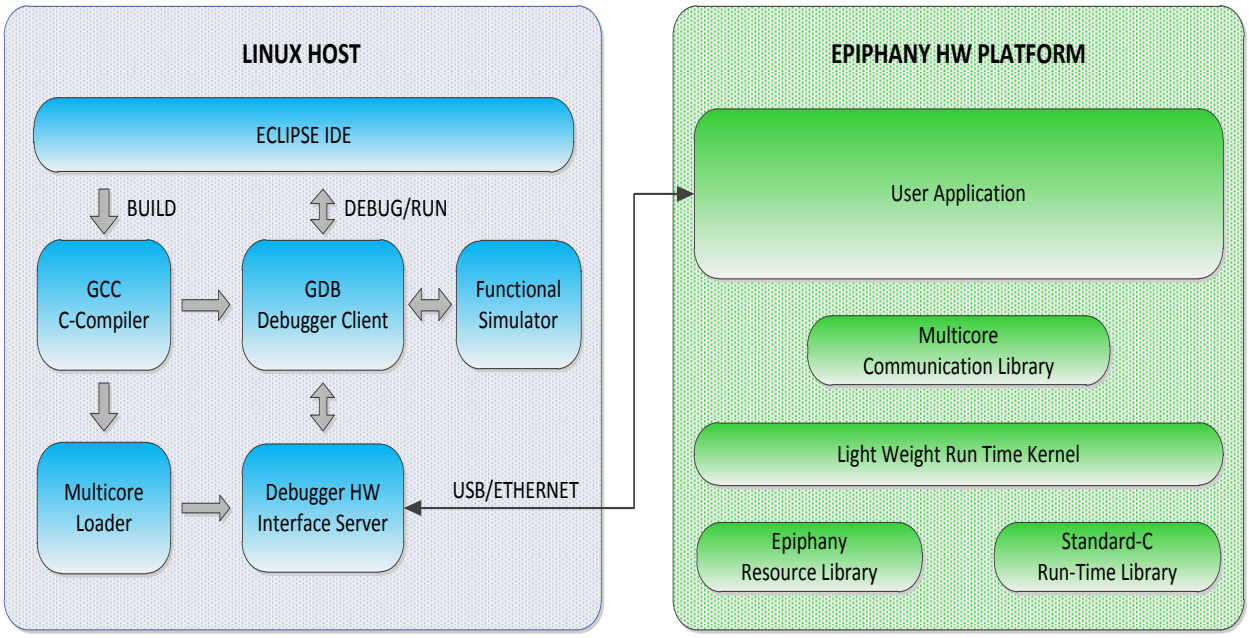
# 3 Software Development Environment

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 compiler
- OpenCL compiler
- Multicore 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 Kit**



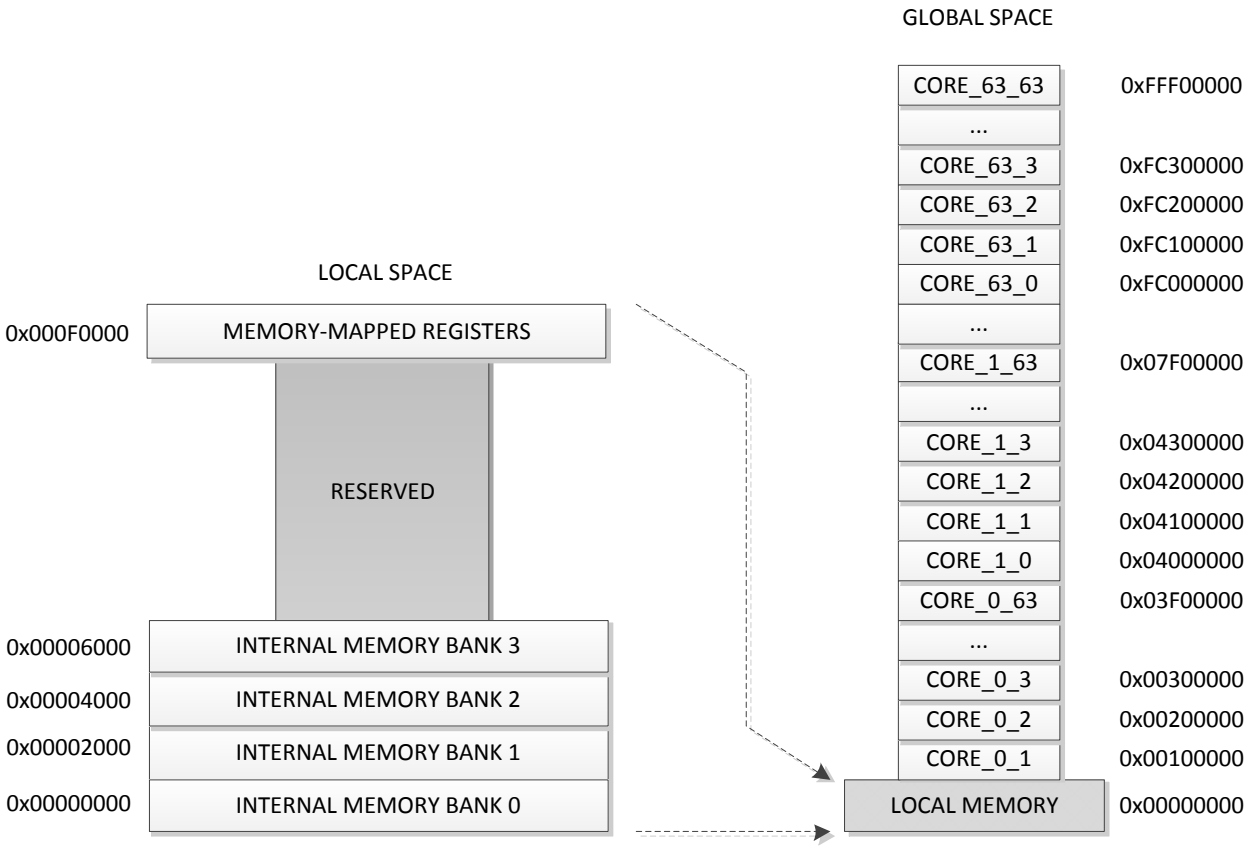

  

# 4 Memory Architecture

## 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**

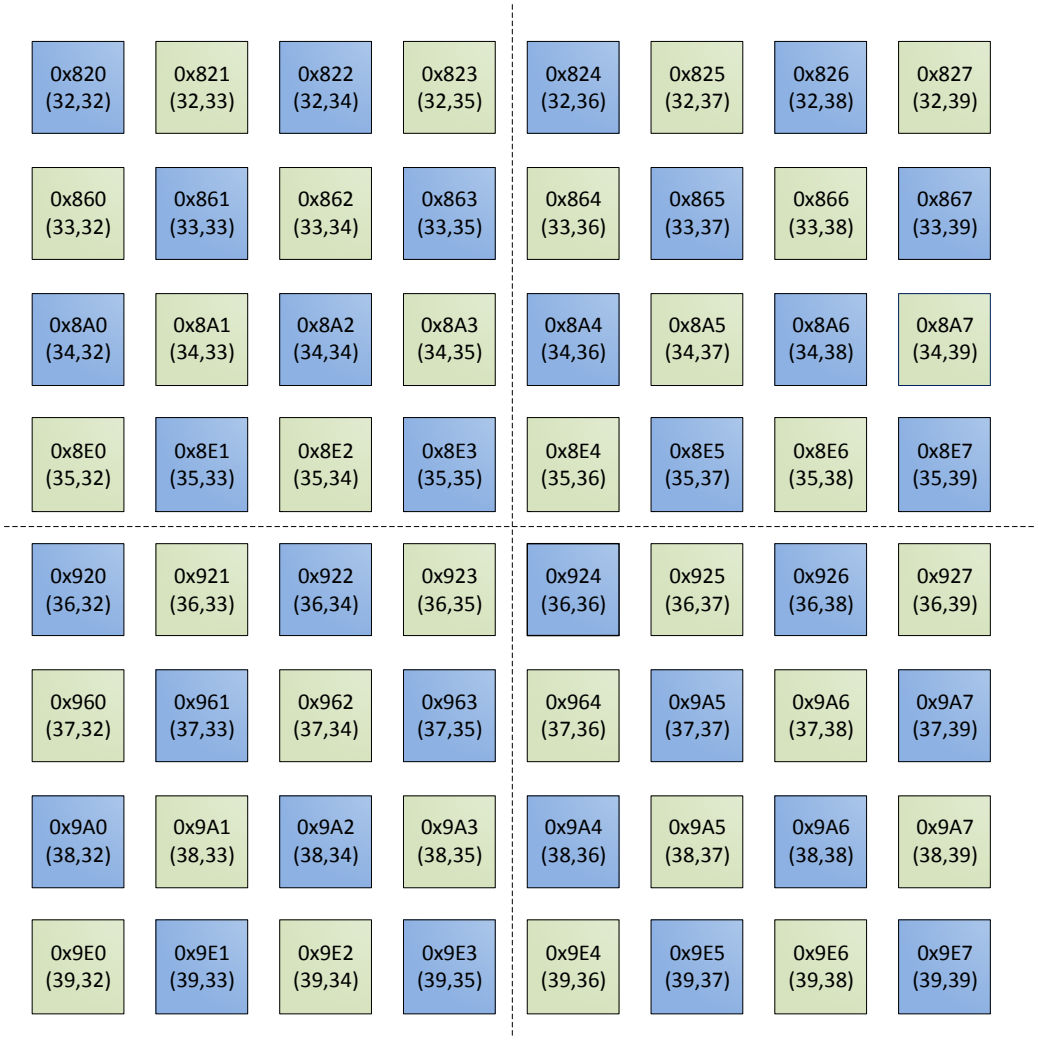


Data and code can be placed anywhere in the memory space or in external space, except for the memory-mapped register space and reserved space, but performance is optimized when the data and code are placed in separate local-memory banks.

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-memory locations, such as off-chip I/O device registers, 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 |
| Write to CoreX | Write to CoreY | No |
| Write to CoreX | Read from CoreY | No |
| Read from CoreX | Write to CoreY | Yes |

## *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 halfword 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 connections. Every router in the mesh is connected to the north, east, west, south, and to a mesh node. 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 (AHB, AXI, etc.). Alternatively, the mesh edge connections can be left unconnected if not needed in the system.

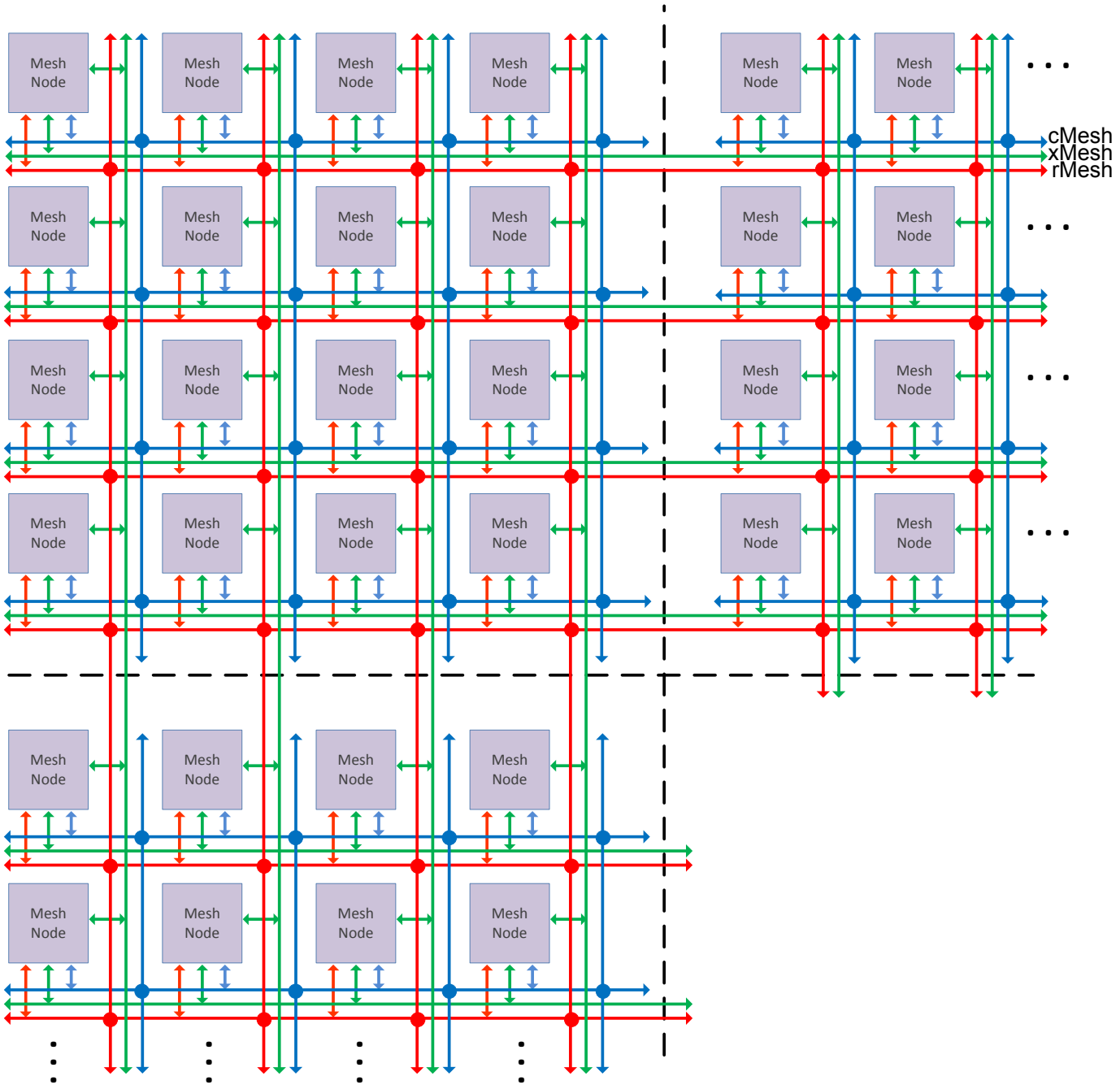
The eMesh Network-on-Chip consists of three separate and orthogonal mesh structures, each serving different types of transaction traffic:

- *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 |

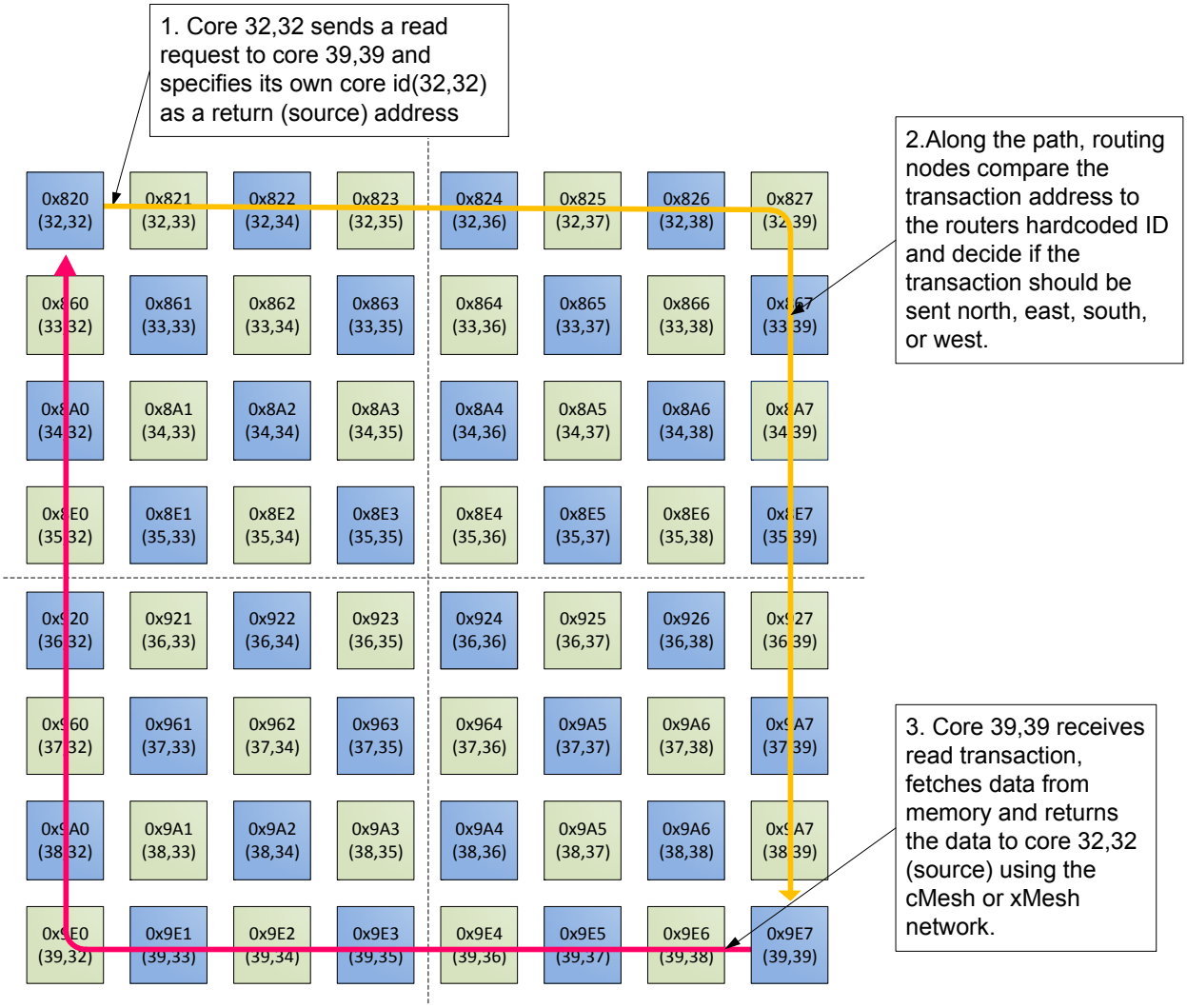
## *5.3 Read Transactions*

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

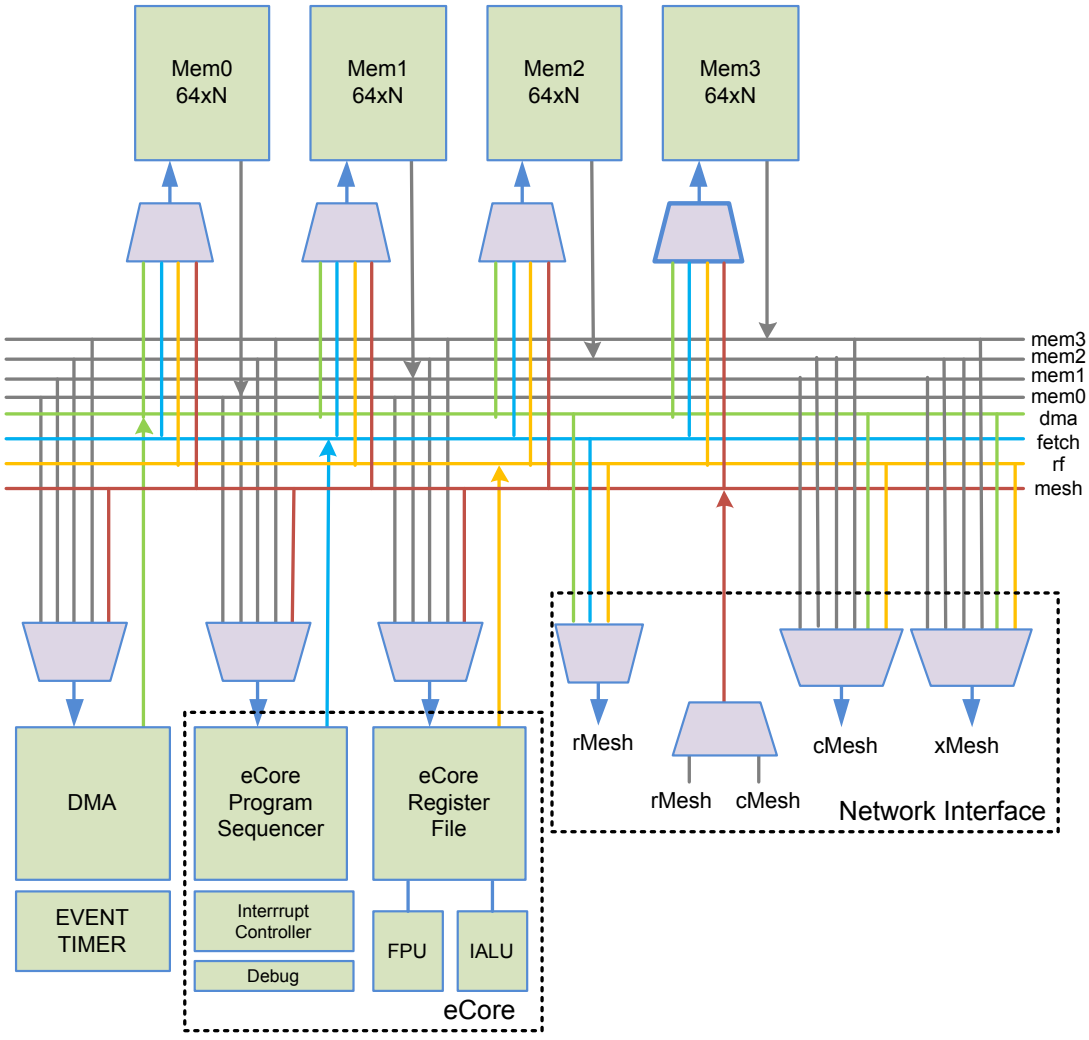
The eMesh network supports byte, halfword, word, or doubleword atomic transactions. Mesh data is always aligned to the least-significant bits (LSBs). Maximum bandwidth is obtained with doubleword transactions. All transactions should have addresses aligned according to the transaction data size.

# 6 Processor Node Subsystem

## 6.1 Processor Node Overview

Figure 11 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**



### 6.1.1 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.

### 6.1.2 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.

### 6.1.3 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 doubleword transaction on every clock cycle.

### 6.1.4 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.

### 6.1.5 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 doubleword write transactions, 8 bytes can be pushed onto the network on every clock cycle without stalling the pipeline.

## 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**

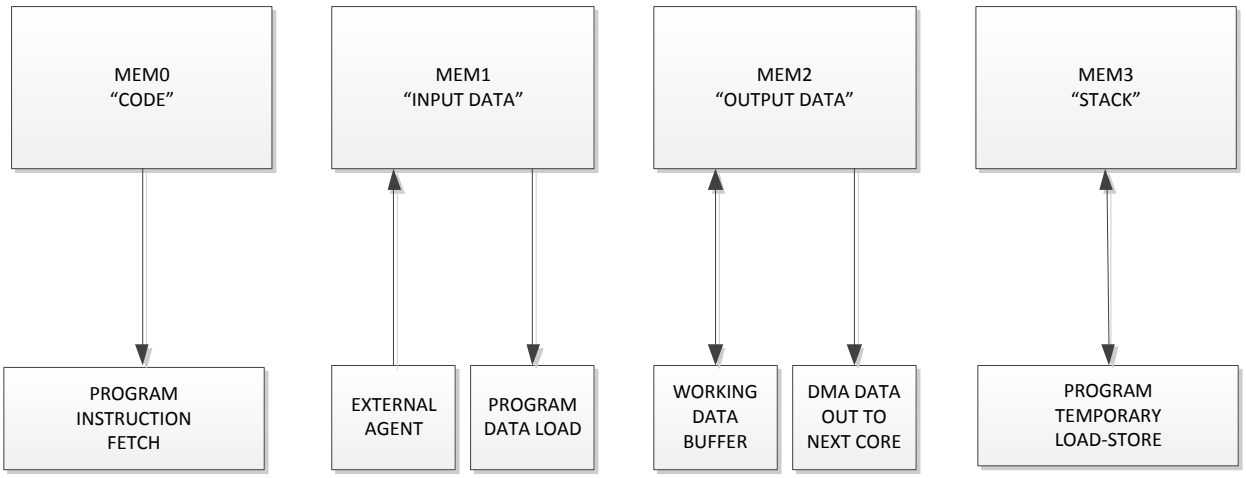
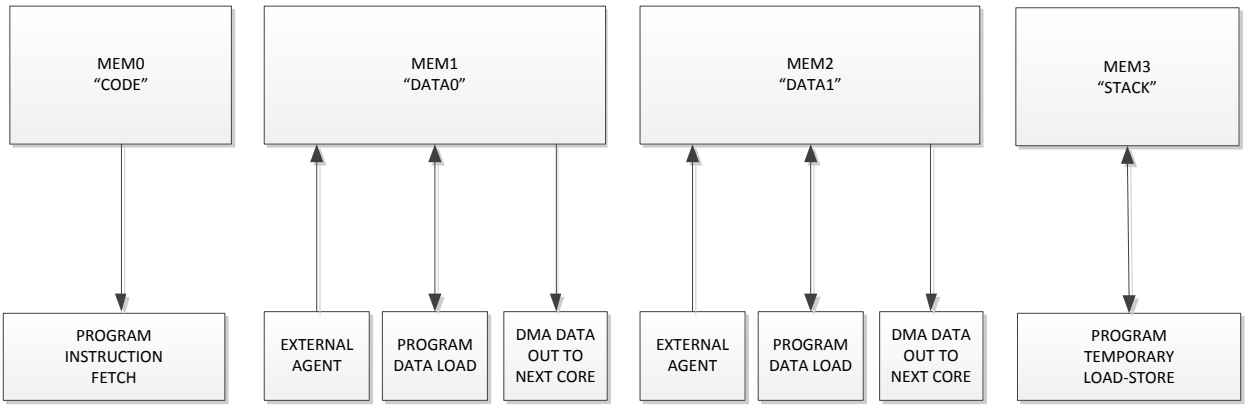



**Figure 12: Program Memory Layout Optimized for Speed**



## 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 |

## 6.4 Mesh-Node ID (COREID)

The row-column coordinate of the processor nodes is contained in the 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 5: Processor node ID Format**

| **COREID (0xF0704)** | | |
|---|---|---|
| **[31:12]** | **[11:6]** | **[5:0]** |
| Reserved | Processor Node Row Coordinate | Processor node Column Coordinate |

## 6.5 Memory Protection Register (MEMPROTECT)

The MEMPROTECT register allows you 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, a memory fault exception is generated. The MEMPROTECT register can be used to help debug program faults related to stack overflow and multicore memory clobbering.

**Table 6: Memory protection register**

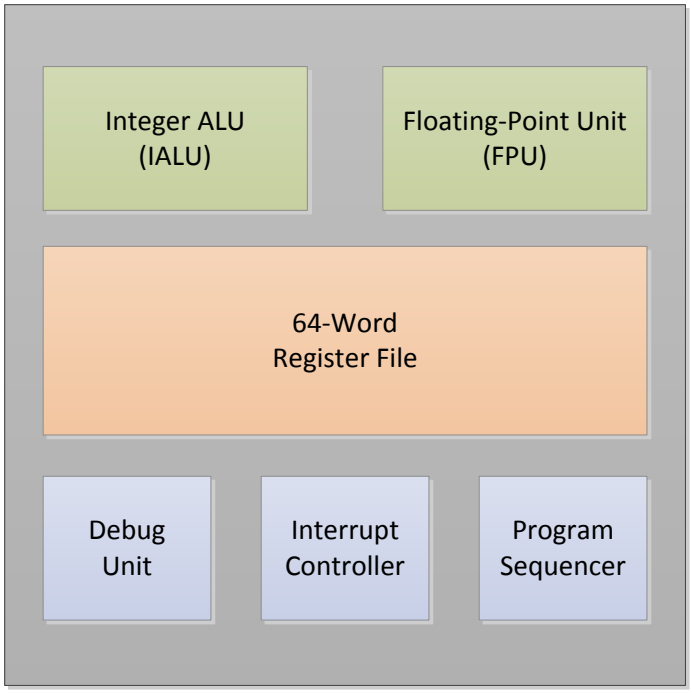
| MEMPROTECT (0xF0704) | |
|---|---|
| [31:8] | [7:0] |
| Reserved | Read only mode bits, one per 4KB page.<br>[0]=1=Addr 0x0000→0x0fff is read-only<br>[1]=1=Addr 0x1000→0x1fff is read-only<br>[2]=1=Addr 0x2000→0x2fff is read-only<br>[3]=1=Addr 0x3000→0x3fff is read-only<br>[4]=1=Addr 0x4000→0x4fff is read-only<br>[5]=1=Addr 0x5000→0x5fff is read-only<br>[6]=1=Addr 0x6000→0x6fff is read-only<br>[7]=1=Addr 0x7000→0x7fff is read-only |

# 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**




### 7.1.1 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.

### 7.1.2 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.

### 7.1.3 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.

### 7.1.4 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.

### 7.1.5 Interrupt Controller

The interrupt controller supports nested interrupts and exceptions for several interrupts sources. The interrupts are prioritized with the following priority order:

- Sync (Highest)
- Software exception
- Memory Protection Fault
- Timer0 Expired
- Timer1 Expired
- DMA Channel 0
- DMA Channel 1
- Software Interrupt (Lowest)

### 7.1.6 Debug Unit

The debug unit works behind the scene with the GNU GDB debugger to 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.

### 7.2.1 Signed Integer Representation

*msb* *lsb*

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

### 7.2.2 Unsigned Integer Representation

*msb* *lsb*

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

### 7.2.3 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 $1 \leq e \leq 254$ 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 7.

**Table 7: 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 8 summarizes the memory map of the eCore CPU local memory.

**Table 8: 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 | Memory expansion |
| Memory Mapped Registers | 0xF0000 | 0xF07FF | 2048 | Mapped register access |
| Reserved | 0xF0800 | 0FFFFF | 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 9. 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 9: 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 eCore Configuration Register

**Table 10: Core Configuration Register**

| Bit | Name | Reset | Function |
|---|---|---|---|
| [0] | RMODE | 0 | IEEE Floating-Point Truncate Rounding Mode<br>0 = Round to nearest even rounding<br>1 = Truncate rounding |
| [1] | IEN | 0 | Invalid floating-point exception enable<br>0 = Exception turned off<br>1 = Exception turned on |
| [2] | OEN | 0 | Overflow floating-point exception enable<br>0 = Exception turned off<br>1 = Exception turned on |
| [3] | UEN | 0 | Underflow floating-point exception enable<br>0 = Exception turned off<br>1 = Exception turned on |
| [7:4] | CTIMER0CFG | 0 | Controls the events counted by CTIMER0.<br>0000 = off<br>0001 = clk<br>0010 = idle cycles<br>0100 = IALU valid instructions<br>0101 = FPU valid instructions<br>0110 = dual issue clock cycles<br>0111 = load (E1) stalls<br>1000 = computational register dependency (RA) stalls<br>1100 = external fetch stalls<br>1101 = external load stalls |
| [11:8] | CTIMER1CFG | 0 | Timer1 mode, same description as for CTIMER0. |
| [16:12] | RESERVED | 0 | N/A |
| [19:17] | ARITHMODE | 0 | Selects the data type used by datapath unit.<br>000 = 32bit float mode<br>100 = 32bit signed integer mode |

| | | | |
|---|---|---|---|
| [31:12] | RESERVED | 0 | N/A |

## 7.6 eCore Status Register

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

**Table 11: Core Status Register**

| Bit | Flag Name | Reset | Updated By | Function |
|---|---|---|---|---|
| [0] | ACTIVE | 0 | Interrupt, IDLE, SWI | Core active indicator<br>0=core idle, 1=core active |
| [1] | GID | 0 | RTI, Interrupt,<br>GIE, GID | Global interrupts disabled indicator<br>0 = all interrupts enabled<br>1 = all interrupts disabled |
| [2] | RESERVED | 0 | N/A | N/A |
| [3] | RESERVED | 0 | N/A | N/A |
| [4] | AZ | 0 | Integer Processing | Integer Zero Flag |
| [5] | AN | 0 | Integer Processing | Integer Negative Flag |
| [6] | AC | 0 | Integer Processing | Integer Carry Flag |
| [7] | AV | 0 | Integer Processing | Integer Overflow Flag |
| [8] | BZ | 0 | Floating-Point | Floating-Point Zero Flag |
| [9] | BN | 0 | Floating-Point | Floating-Point Negative Flag |
| [10] | BV | 0 | Floating-Point | Alternate Overflow Flag |
| [11] | RESERVED | 0 | N/A | N/A |
| [12] | AVS | 0 | Integer Processing | Sticky Integer Overflow |
| [13] | BIS | 0 | Floating-Point | Sticky Floating-Point Invalid |
| [14] | BVS | 0 | Floating-Point | Sticky Floating-Point Overflow |
| [15] | BUS | 0 | Floating-Point | Sticky Floating-Point Underflow |

| | | | | |
|---|---|---|---|---|
| [18:16] | EXCAUSE | 0 | Software exception | Software exception cause |

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 to 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 0,1,2,3 are ready only bits controlled by the operational state of the CPU.

The Core Status Register flags are:

**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.

**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**

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 based exceptions types tracked are (in

binary): 010 = misaligned load/store, 011 = FPU exception, 100 = unimplemented, 000 = no exception.

## 7.7 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. Instructions-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.

### 7.7.1 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 12: Condition Codes**

| Code | Function | Flags | 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 | - |

### 7.7.2 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.

### 7.7.3 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.

### 7.7.4 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.

### 7.7.5 Secondary Signed Integer Instructions

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

### 7.7.6 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.

### 7.7.7 Program Flow Instructions

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

### 7.7.8 Instructions Set Summary

The ISA supports a set of control instructions needed to implement a complete operating system. The following set of tables summarizes the instructions available in the ISA.

**Table 13: 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 14: Branching Instructions**

| Instruction | Assembler | Flags | Function |
|---|---|---|---|
| Conditional Branch | *B<COND> <label>* | None | If <COND>, PC=<label>,<br>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 15: Load/Store Instructions**

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

**Table 16: 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 17: 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-Rm |
| Floating-Point Multiply | *FMUL RD,RN,RM* | BN,BZ,BV, BIS,BVS,BUS | Rd=Rn*Rm |
| Floating-Point Fused Multiply Add | *FMADD RD,RN,RM* | BN,BZ,BV, BIS,BVS,BUS | Rd+=Rn*Rm |
| Floating-Point Fused 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 18: 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 |

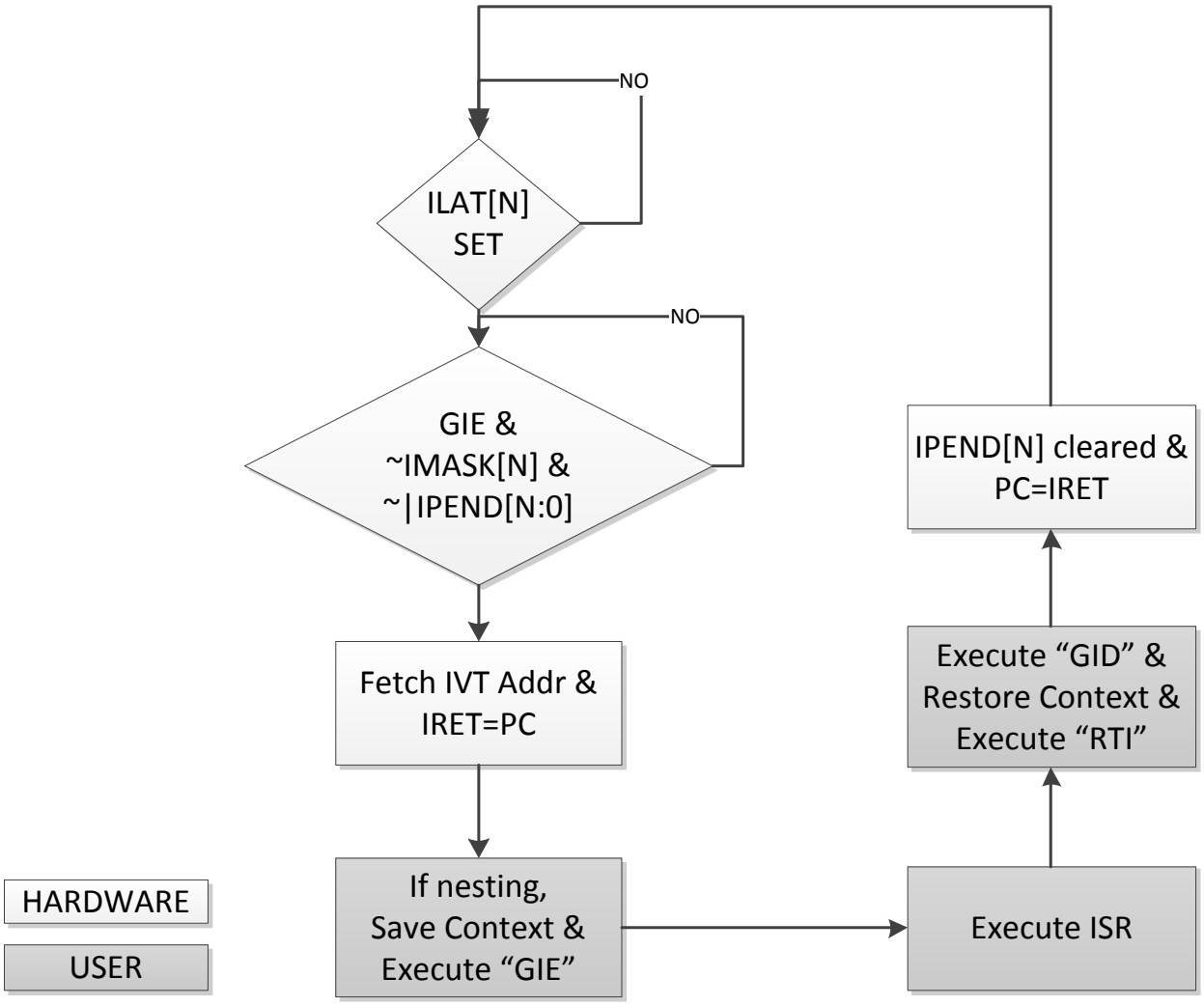
**Table 19: Program Flow Instructions**

| Instruction | Assembler | Flags | 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 |
| Trap | *TRAP* | None | Halts program |

## 7.8 Interrupt and Exception Handling

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 14: 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 20: Interrupt Support Summary**

| Interrupt/Exception | 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 |
| Reserved | 5 | 0x14 | N/A |
| DMA0 Interrupt | 6 | 0x18 | Local DMA channel-0 finished data transfer |
| DMA1 Interrupt | 7 | 0x1C | Local DMA channel-1 finished data transfer |
| Reserved | 8 | 0x20 | N/A |
| Software Interrupt | 9 (lowest) | 0x24 | Software generated |

The IVT contains a set of 32-bit relative branch instructions that point to the appropriate interrupt handlers. The interrupt handlers must be placed in the core's local memory. The following Figure shows the execution flow of the interrupt control engine and how the eCore provides fine grained control over the execution of nested interrupt service routines.

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 7 in the ILAT register.

### 7.8.1 Interrupt Routine Link (IRET)

The program counter of the instruction following the one currently being executed is saved in the IRET register when an interrupt starts being serviced. 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.

### 7.8.2 Interrupt Mask (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. A value of 1 in a specific bit of the IMASK register means the interrupt is blocked from occurring. An interrupt is still latched by the ILAT register when the corresponding IMASK bit is set, but it will be blocked from affecting the program sequencer execution flow until the IMASK bit is cleared to zero.

### 7.8.3 Interrupt Latch (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. The ILAT register should not be written to directly by software, as the write could interfere with hardware interrupts setting their appropriate bits in the ILAT register. The correct method of accessing individual bits of the ILAT register is by using the ILATST and ILATCL register aliases described immediately below.

### 7.8.4 Interrupt Latch Set Alias (ILATST)

An alias for the ILAT register that allows bits within the ILAT register to be set individually. Writing a one 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 register cannot be read.

### 7.8.5 Interrupt Latch Clear Alias (ILATCL)

An alias for the ILAT register that allows bits within the ILAT register to be cleared individually. Writing a one 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 register cannot be read.

### 7.8.6 Interrupt Service Pending Status (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.

### 7.8.7 Status Register (STATUS)

Bit [1] of the Core 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.

### 7.8.8 Global Enabling/Disabling of Interrupts

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.

### 7.8.9 Software 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.

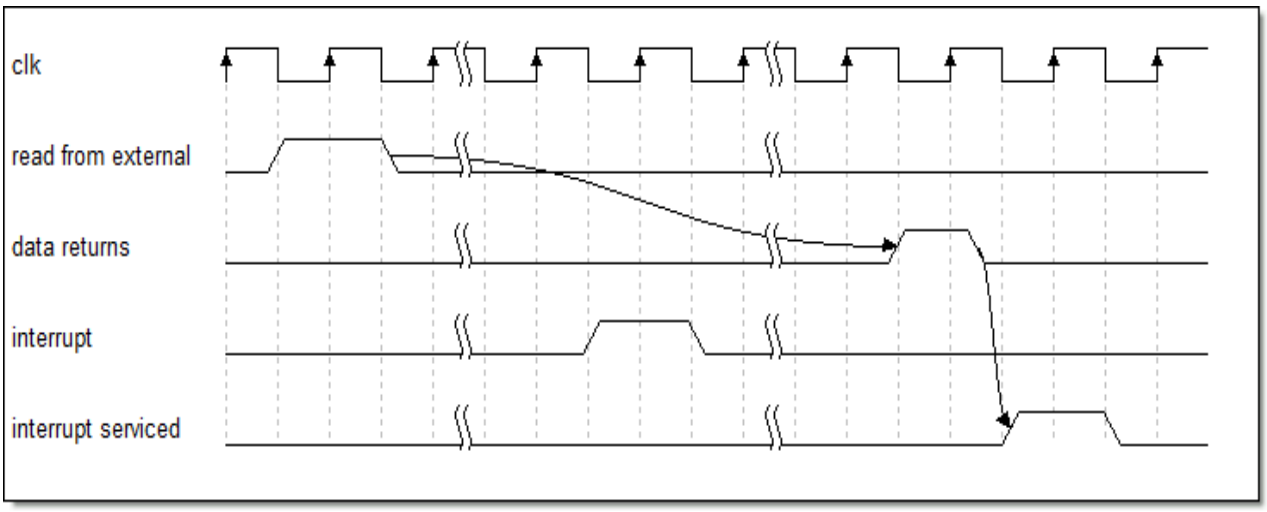
To minimize interrupt latency, two requirements should be met:

- 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 15: Interrupt Latency Dependency On External Read**



## 7.9 Pipeline Description

The ISA 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.

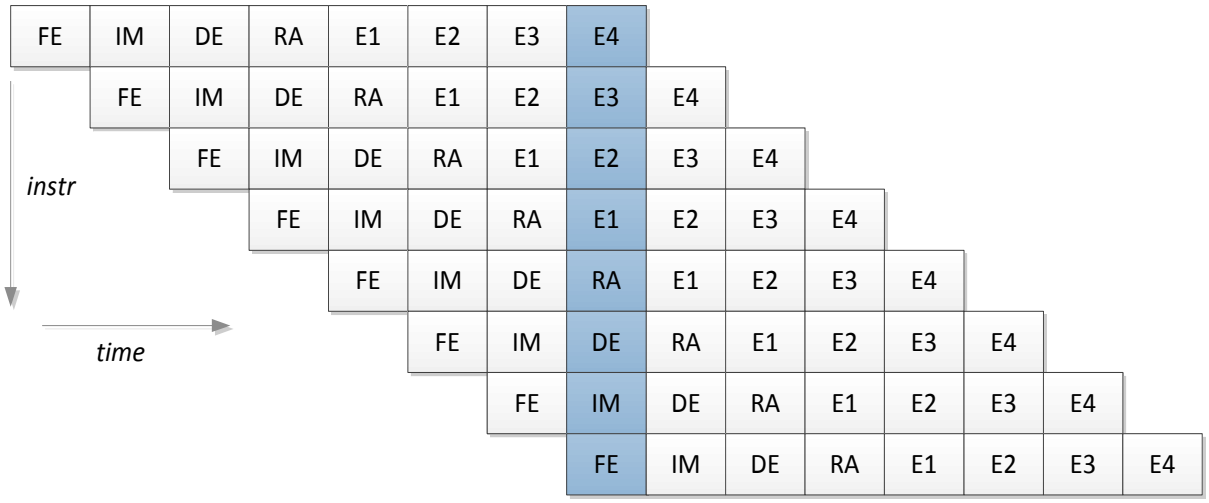
Instructions are dispatched in-order but can finish out-of-order. The pipeline controller makes sure that the integrity of the program is maintained by stalling the pipeline appropriately if there is a read-after-write (RAW) or write-after-write (WAW) pipeline hazard.

**Table 21: Pipeline Stage Description**

| Stage | Name | Mnemonic | Action |
|---|---|---|---|
| 1 | Fetch Address | FE | Fetch address sent to instruction memory |
| 2 | Instruction Returns | 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<br>Register read from register file for memory store operation<br>Most instructions completed<br>Integer status flags written<br>Branching and jumps change program flow |
| 6 | Execution | E2 | Data from load instruction written to register file |
| 8 | Execution | E3 | Floating-point execution cycle |
| 9 | Execution | E4 | Floating-point result written to register file and floating-point status flags updated |

**Figure 16: 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.

## 7.10 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 | and | eor | and | orr | bitr | movt | mov<cond>
- FPU: fadd | fsub | fmul | fmadd | fmsub | fix | float
- Load: ldr<size>
- Store: str<size>
- Branch: b<cond> | bl
- Register jump: jr | jalr
- Special move: movts | movfs
- Control: Register jump | Branch | Special move

The following table shows the combinations of instructions that can be executed in parallel.

**Table 22: Parallel scheduling rules**

| Instruction Type | IALU | FPU | Load/Store | Control |
|---|---|---|---|---|
| IALU | | YES | NO | NO |
| FPU | 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 23: IALU Instruction Sequences**

| First Instruction | Second Instruction | Cycle Separation |
|---|---|---|
| IALU Instruction<br>*mov rx, rz, rz* | FPU Instruction<br>*fadd rx, rx, rx* | 1 |

**Table 24: FPU Instruction Sequences**

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

**Table 25: Load Instruction Sequences**

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

**Table 26: Stalls independent of Instruction Sequence**

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

## 7.11 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 27: 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.

# 8 Direct Memory Access (DMA)

## 8.1 Overview

Each Epiphany processor node contains a DMA engine to offload data movement across the eMesh network. The DMA engine works at the same clock frequency as the CPU and can transfer a 64-bit doubleword 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 and destination address configuration per descriptor and channel.
- 2D DMA operation.
- Flexible stride sizes.
- DMA descriptor chaining.
- Hardware interrupt flagging to local CPU subsystem.

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

**Table 28: 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 Configuration Register (DMA{0,1}CONFIG)

The DMA config 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 29: DMA Configuration Register**

| Bit | Name | Reset Value | Function |
|---|---|---|---|
| [0] | DMAEN | 0 | Turns on DMA channel.<br>1=enabled<br>0=disabled |
| [1] | MASTER | 0 | Sets up DMA channel to work in master made<br>1=master mode<br>0=slave mode |
| [2] | CHAINMODE | 0 | Sets up DMA in chaining mode, meaning that a new descriptor is fetched from the next descriptor address at the end of the current configuration.<br>1=Chain mode<br>0=One-shot mode |
| [3] | STARTUP | 0 | 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.<br>1=Fetch descriptor<br>0=Normal operation |
| [4] | IRQEN | 0 | 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.<br>1=Enable interrupt at end of DMA transfer<br>0=Disable interrupt at end of DMA transfer. |
| [6:5] | DATASIZE | 0 | Size of data transfer.<br>00=byte,01=halfword.10=word. 11=doubleword |
| [15:7] | RESERVED | 0 | N/A |
| [31:16] | NEXT_PTR | 0 | Address of next DMA descriptor for normal operation.<br>Address of immediate descriptor to fetch in case of startup mode. |

## 8.3 Count Register (DMA{0,1}COUNT)

This register is used to set up the number of transactions in the inner and outer loop 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 should always be set to at least one at the start to ensure correct operation. The DMA block transfer is done when the complete DMACOUNT register reaches zero. The inner count value is cleared to the initial count every time the outer loop is decremented.

## 8.4 Stride Register (DMA{0,1}STRIDE)

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 the stride to update the source address register and the upper 16 bits specify the stride used to update the destination address register. At the end of an inner-loop turn, this register is loaded with the outer-loop stride values to perform a 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.

## 8.5 Source Address Register (DMA{0,1}SRCADDR)

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.

## 8.6 Destination Address Register (DMA{0,1}DSTADDR)

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.

## 8.7 AUTODMA Register (DMA{0,1}AUTO0 / DMA{0,1}AUTO1)

The AUTODMA register is a streaming channel end point that can be written to by an external agent within the system when the DMA channel is configured in slave mode. In slave mode, the rate of transactions is controlled by the rate of writes being done to the AUTODMA register rather than by the DMA transaction sequencer. The AUTODMA0/1 makes a pair that can receive a doubleword transaction. The AUTODMA register can be used to set up efficient streaming channels without having to manage addresses of the destination processor.

## 8.8 Status Register (DMA{0,1}STATUS)

The DMA status register contains the state of the DMA channel, as shown in the table below.

**Table 30: DMA Status Register**

| Bit | Reset Value | Name | Function |
|---|---|---|---|
| [3:0] | 0 | DMASTATE | 0x0=DMA idle<br>0x5=DMA actively processing transaction<br>0x6=DMA in slave mode waiting for transaction<br>0xD=A DMA configuration error. Occurs when the DMA configuration register is written while |

| | | | |
|---|---|---|---|
| | | | the DMA is not in in an IDLE state.<br>All other DMA states are temporary in nature and are not meaningful to the user. |
| [15:4] | 0 | RESERVED | N/A |
| [31:16] | 0 | CURR_PTR | The address of DMA descriptor currently being processed. |

## 8.9 DMA Descriptors

The format of DMA descriptors is given in the table below. All descriptors should be placed in local memory and must be doubleword 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 31: 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) |

## 8.10 DMA Channel Arbitration

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

## 8.11 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.

## 8.12 DMA Transfer Example

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

```
MOV   R1, 0x8          ; set the startup bit
MOVT  R1, _DESCRIPTOR0 ; put descriptor pointer in the upper 16 bits
MOVTS DMA0CONFIG, R1   ; start a DMA transfer by writing to the
                       ; DMA config register.

_DESCRIPTOR0;
.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 in which two instructions are 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 the 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.

## *9.1 CTIMER0*

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.

## *9.2 CTIMER1*

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.

# 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
```

## *BITR*

**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 (UbiasedExponent(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

BV Flag

BVS Flag

**Operation:** RD=RN + RM

BN = RD[31]

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

BV = 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

BV Flag

BIS 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

BV 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

BV 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

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}

**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(!INTERRUPTED){
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/8)-1);

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/8)-1);;

**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  ;
```

## *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 | SIMM8) 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/8)-1);

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/8)-1)

**Example:**

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

## *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. Standard I/O is implemented as part of the Epiphany SDK, so there should not be a need to use the TRAP instruction within user software.

**Syntax:** TRAP <IMM3>

<IMM3> An unsigned 3 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 within an 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
```

# Appendix B: Register Description Reference

This appendix contains reference tables for all the registers within the Epiphany architecture. To access these registers using a memory read or write transaction, set the upper twelve bits of the 32-bit address to the appropriate processor node ID containing the register in question. For example, to read the status register of core (32,32) you would specify an address of 0x820F0404.

**Table 32: eCore Registers**

| Address | Register Name | Reset Value | Comment |
|---|---|---|---|
| 0xF0000→ 0xF00FC | R0-R63 | N/A | General purpose registers |
| 0xF0400 | CONFIG | 0x0 | Core Configuration |
| 0xF0404 | STATUS | 0x0 | Core Status |
| 0xF0408 | PC | 0x0 | Program Counter |
| 0xF0420 | IRET | 0x0 | Interrupt PC return register |
| 0xF0424 | IMASK | 0x0 | Interrupt masking register |
| 0xF0428 | ILAT | 0x0 | Interrupt latch register |
| 0xF042C | ILATST | 0x0 | Alias for setting interrupts |
| 0xF0430 | ILATCL | 0x0 | Alias for clearing interrupts |
| 0xF0434 | IPEND | 0x0 | Interrupts currently in process |

**Table 33: DMA Registers**

| Address | Register Name | Reset Value | Address |
|---|---|---|---|
| 0xF0500 | DMA0CONFIG | 0x0 | 0xF0500 |
| 0xF0504 | DMA0STRIDE | 0x0 | 0xF0504 |
| 0xF0508 | DMA0COUNT | 0x0 | 0xF0508 |
| 0xF050C | DMA0SRCADDR | 0x0 | 0xF050C |
| 0xF0510 | DMA0DSTADDR | 0x0 | 0xF0510 |
| 0xF0514 | DMA0AUTO0 | 0x0 | 0xF0514 |
| 0xF0518 | DMA0AUTO1 | 0x0 | 0xF0518 |
| 0xF051C | DMA0STATUS | 0x0 | 0xF051C |
| 0xF0520 | DMA1CONFIG | 0x0 | 0xF0520 |
| 0xF0524 | DMA1STRIDE | 0x0 | 0xF0524 |
| 0xF0528 | DMA1COUNT | 0x0 | 0xF0528 |
| 0xF052C | DMA1SRCADDR | 0x0 | 0xF052C |
| 0xF0530 | DMA1DSTADDR | 0x0 | 0xF0530 |
| 0xF0534 | DMA1AUTO0 | 0x0 | 0xF0534 |
| 0xF0538 | DMA1AUTO1 | 0x0 | 0xF0538 |
| 0xF053C | DMA1STATUS | 0x0 | 0xF053C |

**Table 34: Event Timer Registers**

| Address | Register Name | Reset Value | Comment |
|---|---|---|---|
| 0xF0438 | CTIMER0 | 0x0 | Core Timer0 |
| 0xF043C | CTIMER1 | 0x0 | Core Timer1 |

**Table 35: Processor Control Registers**

| Address | Register Name | Reset Value | Comment |
|---|---|---|---|
| 0xF0608 | MEMPROTECT | 0x0 | Processor node Memory Protection |
| 0xF0704 | COREID | 0x0 | Processor node Core ID |