

A scalable manycore simulator for the Epiphany architecture

Master's thesis in Computer science and engineering

Ola Jeppsson

MASTER'S THESIS 2019

A scalable manycore simulator for the Epiphany architecture

Ola Jeppsson



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

A scalable manycore simulator for the Epiphany architecture
Ola Jeppsson

© Ola Jeppsson, 2019.

Supervisor: Sally A. McKee, Department of Computer Science and Engineering
Examiner: Mary Sheeran, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in L^AT_EX
Gothenburg, Sweden 2019

A scalable manycore simulator for the Epiphany architecture

Ola Jeppsson

Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

Abstract

The core count of manycore processors increases at a rapid pace; chips with hundreds of cores are readily available, and thousands of cores on a single die have been demonstrated. A scalable model is needed to be able to effectively simulate this class of processors. We implement a parallel functional network-on-chip simulator for the Adapteva Epiphany architecture, which we integrate with an existing single-core simulator to create a manycore model. To verify the implementation, we run a set of example programs from the Epiphany SDK and the Epiphany toolchain test suite against the simulator. We run a parallel matrix multiplication program against the simulator spread across a varying number of networked computing nodes to verify the MPI implementation. Having a manycore simulator makes it possible to develop and optimize scalable applications even before the chips for which they are designed become available. The simulator can also be used for parameter selection when exploring richer hardware design spaces.

Keywords: Simulation, Functional Simulator, Manycore, Network-on-Chip, Adapteva, Epiphany, eMesh, Shared Memory, MPI, Process-level parallelism, GDB

Preface

The Epiphany mesh network-on-chip simulator described in this thesis is original, independent work by the author. A shorter description has previously been published in Jeppsson and McKee [1]. This thesis is based on the Epiphany port of the GDB (The GNU Debugger) simulator framework. GDB is a product of the Free Software Foundation.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my thesis supervisor Professor Sally A. McKee. Prof. McKee has been a great mentor and has been nothing but encouraging throughout the writing of this thesis. I would also like to thank my thesis examiner Professor Mary Sheeran for her patience, vital feedback, and sincere interest in making the thesis better. In addition, thank you to Andreas Olofsson of Adapteva Inc. Mr. Olofsson has always been available to answer any questions about architecture details. This has been of utmost importance for the outcome of this thesis. Last but not least, I would like to thank my parents and brother for all their support and for always being there for me.

Ola Jeppsson, Clemson, August 2019

Contents

List of Figures	xi
------------------------	-----------

List of Tables	xiii
-----------------------	-------------

1 Introduction	1
1.1 Why a manycore simulator is needed	1
1.2 Method	2
1.3 Limitations	2
1.4 Related work	3
1.5 Document structure	3
2 Background	5
2.1 Epiphany architecture	5
2.1.1 RISC cores	5
2.1.2 Network-on-Chip	6
2.1.3 Addressing scheme	6
2.1.4 Core memory region layout	7
2.1.5 Memory model	7
2.2 The Parallella board	8
2.3 The GNU debugger	9
2.3.1 Simulator framework	10
3 Implementation	11
3.1 Single-core simulator integration	12
3.1.1 Interrupt implementation	12
3.1.2 Hardware loops	14
3.2 Device implementation	14
3.2.1 Timers	14
3.2.2 DMA	15
3.3 eMesh simulator	15
3.4 Parallella/Epiphany software development kit (SDK) support	15
3.5 Networking support	16
3.6 Simulator frontend	16
3.7 Simulation environments	17
4 Results	19
4.1 Epiphany examples	19

4.2	Domino	21
4.3	HPC demonstration	21
4.4	Simulator frontend	24
4.5	Integration with Epiphany SDK	24
5	Discussion	25
5.1	Future work	26
5.1.1	Clock count estimation	26
5.1.2	Multicore debugging support	27
6	Conclusion	29
	Bibliography	31
	Acronyms	33
	Appendices	35
A	Simulator frontend usage	37

List of Figures

2.1	Epiphany architecture	6
2.2	Epiphany address layout	6
2.3	Epiphany core address map	7
2.4	Parallella board	8
2.5	Parallella functional diagram	9
2.6	GDB overview	9
3.1	Simulator overview	11
3.2	Interrupt service routine operation	14
3.3	Host device communication	16
3.4	Simulator frontend overview	17
4.1	epiphany-examples test suite	20
4.2	Domino message path	21
5.1	NoC delay formula	26

List of Tables

2.1	Memory ordering guarantees	8
4.1	Legend description for tables 4.2 to 4.5	23
4.2	Results for matmul-16	23
4.3	Results for matmul-64	23
4.4	Results for matmul-256	24
4.5	Results for matmul-1024*	24

1

Introduction

For many decades shrinking feature sizes, following Moore’s law, have enabled more and more transistors on-chip. This has allowed processor architects to consistently deliver higher-performing designs by raising clock speeds and adding more hardware to increase instruction-level parallelism (ILP). Techniques like speculative and out-of-order execution not only increase performance — they also increase power consumption, which, in turn, increases heat dissipation.

Power and thermal considerations are now first-class design parameters. Since the performance gains from adding more complexity to a micro-architecture is only square root proportional to the increase in power consumption [2], ILP does not scale as well as thread-level parallelism in that regard. In the previous decade, industry leaders turned to chip multiprocessor designs to deliver more parallel performance instead of focusing on single-threaded performance. Some processor manufacturers have since turned to manycore designs to deliver even greater parallel performance. Chips with hundreds of cores are readily available today [3, 4]. The *KiloCore* [5] project has demonstrated a functioning 1000-core chip. In 2016 Adapteva taped out *Epiphany-V* [6], a 1024-core chip.

The Adapteva Epiphany architecture [7] is a manycore processor design consisting of reduced instruction set computing (RISC) cores organized in a 2D mesh network topology. This type of 2D grid layout is a popular topology for manycore architectures due to its relative ease of design and scalability. The cores are connected by three network-on-chips (NoCs) for different types of traffic. The programming model offers a shared address space in which every core has a dedicated memory region. Chips with 16-64 cores are readily available, but the architecture is designed to accommodate thousands of cores per chip, and the address space supports up to 4095 cores¹ in one shared address space. Section 2.1 provides a more detailed architecture description.

1.1 Why a manycore simulator is needed

Prior to this thesis there existed only a single-core simulator for the Epiphany architecture: simulating an Epiphany chip with all cores running concurrently was not possible. While a single-core simulator is useful for tasks like register-transfer level (RTL) code verification, and running a toolchain’s (compiler, assembler, linker, debugger) test suite, most real-world applications targeting manycore systems naturally are multiple instruction streams, multiple data streams (MIMD).

¹Epiphany-V 64-bit architecture supports systems with up to 1 billion cores.

Further, a manycore simulator makes it possible to explore richer hardware design spaces and makes it possible to develop and optimize scalable applications (even before the chips for which they are designed become available).

A working Epiphany manycore simulator is a vital building block for doing full-system simulation of the Parallella microcomputer board [8]. See section 2.2 for a description of the Parallella board.

1.2 Method

The objective of this Master’s thesis is to create a manycore simulator for the Epiphany architecture. We do this by implementing a functional simulator for the Epiphany eMesh NoC, see section 3.3. The eMesh simulator is integrated with the existing single-core simulator to create a manycore simulator; this is described in section 3.1. We extend the single-core simulator previously missing features, such as support for interrupts and self-modifying code.

To exploit maximum parallelism in the simulated applications, we add support for distributing the simulation across several networked nodes via message passing interface (MPI). MPI is commonly supported in smaller clusters as well as in high-performance computing (HPC) environments.

To verify the implementation, we run a set of example programs from the Epiphany SDK and the Epiphany toolchain test suite against the simulator. We run a parallel matrix multiplication program against the simulator spread across a varying number of networked computing nodes to verify the MPI implementation.

1.3 Limitations

The focus of this thesis is on developing a functioning and scalable manycore simulator for the Epiphany architecture. Features outside the thesis’s main scope have not been prioritized and should therefore not be taken into account when assessing the outcome of the project. These include:

- core functionality not implemented in the single-core simulator, i.e., functionality not dependent on the NoC;
- cycle-accurate timing;
- detailed simulation of the on-chip network, i.e., instead of routing memory traffic via intermediate nodes, we send requests directly to the target core;
- simulation of other components on the Parallella board (the Epiphany reference platform). This includes details of the ARM cores, Zynq functionality, and operating system (OS) activity;
- features marked as *LABS* in the reference manual [9] (i.e., untested or broken functionalities);
- *Epiphany-V* features. These include 64-bit addressing mode, double-precision floating-point support, and new instruction set architecture (ISA) instructions.

The reasons for why we do not implement these features are the limited time at our disposal, and that we want the simulator to be as fast as possible.

1.4 Related work

There exists a variety of different CPU simulators having different implementations and different goals. Some offer very precise simulation while others barely emulate the target to achieve as close to native execution speed as possible. Before we started working on the simulator, we compared different simulator frameworks, both to get inspiration and to see if any of them was more suitable than the GDB simulator framework the already available single-core Epiphany simulator was using.

- QEMU [10] is a widely used machine emulator. QEMU supports both user mode and full-system simulation. It has support for a wide variety of CPU architectures and peripherals. QEMU uses dynamic code translation to increase execution speed. It can simulate a multi-processor system but does so in a single thread. There is also ongoing work to support multi-threaded code translation [11], but this work has not yet been merged into mainline QEMU. PQEMU [12] is a QEMU fork that uses multiple threads to speed up simulation.
- gem5 [13] is a competent full-system simulator. Work to implement multiple event queues (for parallelism) had just begun when this project started and was not ready for use. gem5 has its own domain-specific language (DSL) for describing ISA opcodes and semantics, which should make it easier to add new architectures. gem5 can be configured to provide a fine-grained, timing-accurate simulation. gem5 can support both user-mode and full-system simulation.
- SID [14] is another computer system simulation framework. It can be used to simulate complex computer systems with multiple cores, buses, and devices. It is relevant for this work because of its CGEN (see section 2.3) support. CGEN is used by the existing single-core simulator, which means that the ISA opcodes and semantics description could be easily reused.
- Parallel Embra [15] is a functional multicore simulator. It uses OS-level threads for parallelism and uses the underlying memory system of the host to synchronize memory accesses. Parallel Embra supports full-system simulation. To speed simulation, it uses binary translation and loose, but functionally correct, timing constraints.
- Graphite [16] is the only CPU simulator we considered that supports distributing a simulation across multiple machines. It does not use MPI, which is the goto programming model in the HPC world, but instead rolls its own implementation, which promises to be more dynamic and to make it more easy for users since they do not need to preallocate resources.

1.5 Document structure

The rest of this thesis is organized as follows. Chapter 2 introduces the background and relevant concepts needed to understand the rest of the thesis. Chapter 3 describes the implementation. Chapter 4 shows the results and verification. Chapter 5 discusses the results and further work. Chapter 6 provides a summary of the thesis outcome.

2

Background

This chapter introduces concepts relevant to Adapteva technology and the GDB simulator framework. The information in this chapter is needed to understand and implement a functional Epiphany manycore simulator. The actual implementation is described in chapter 3.

2.1 Epiphany architecture

The Epiphany architecture is a manycore processor design comprised of three main components: cores, NoCs, and off-chip I/O (eLink). These are depicted in fig. 2.1. The cores are placed in a two-dimensional grid topology. They are connected by three separate on-chip networks for different types of traffic, with one router per intersection.

Both the cores and the NoCs have been designed for scalability and energy efficiency, and therefore many things one would expect in standard multicore processors have been stripped away. For instance, there are no inter-core buses, no caches (and hence no cache-coherence protocol), and no speculative execution (only static branch prediction). The upside of these trade-offs is an energy-efficient architecture that can fit thousands of cores on a single die [6].

2.1.1 RISC cores

The cores are simple RISCs, each equipped with one integer arithmetic logic unit (ALU) and one IEEE-754 [17] compatible single-precision floating-point unit (FPU). The FPU can also be configured to operate as a second ALU. The pipeline is dual-issue for the ALU and FPU data paths. Alternatively, it can also execute one FPU instruction and one load/store instruction per clock cycle. Since the FPU has fused multiply-accumulate instructions, a core can effectively execute two floating-point operations and one load/store operation per clock cycle. The ISA includes about 40 instructions. The register file has 64 general-purpose registers and a group of special-core registers. All registers are memory-mapped. The basic word-size is 32 bits. There are no caches or memory management unit (MMU), so the memory needs to be explicitly managed by the programmer. Branch prediction is static non-taken only, but to compensate for that there are hardware loops. Each core also has two event timers, a two-channel direct memory access (DMA) unit, and an interrupt controller with nested interrupt support.

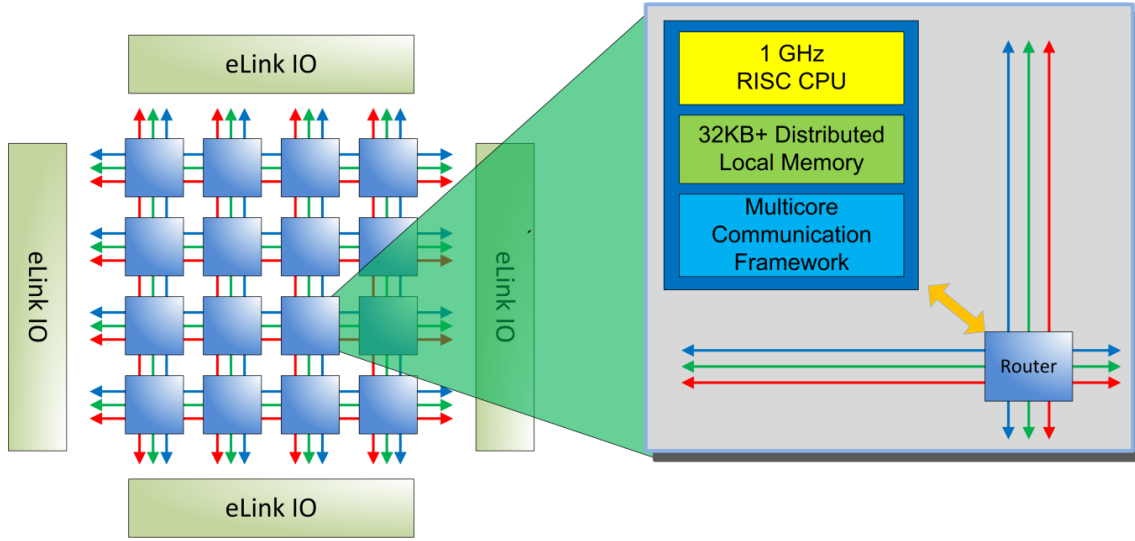


Figure 2.1: Epiphany architecture [9]

2.1.2 Network-on-Chip

Network-on-chip is a network-based approach for communicating between different components on an integrated circuit (IC). It uses router-based packet-switching for communication. Compared to traditional solutions such as system buses and crossbars, it offers better scalability traded for higher latency.

Figure 2.1 shows the 2D mesh layout of the Epiphany architecture NoC. The Epiphany implements separate networks for different types of traffic: one for reads (the *rMesh*), one for on-chip writes (the *cMesh*), and one for off-chip writes (the *xMesh*). We collectively refer to these as the *eMesh*. Packets are first routed east-west and then north-south. A chip has one *eLink* per side. At a clock frequency of 1 GHz, total off-chip bandwidth is 6.4 GB/s, and total on-chip network bandwidth is 64 GB/s at every core router.

2.1.3 Addressing scheme

The architecture has a shared, globally addressable 32-bit address space. An address is logically divided into *coreid* (upper 12 bits), and *offset* (lower 20 bits). The *coreid* determines to which core a memory access should be routed. Addresses with all *coreid* bits set to zero alias to the local core's memory region. Further, the *coreid* is divided into *row* (upper six bits), and *column* (lower six bits). Thus the addressing scheme can support a grid of up to 64×64 cores, each with its own 1 MB memory region. This is depicted in fig. 2.2.

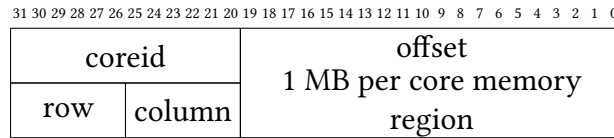


Figure 2.2: Epiphany address layout

2.1.4 Core memory region layout

The core memory layout for the Epiphany-III is shown in fig. 2.3. Memory in the range $[0x0000-0x8000]$ is backed by standard SRAM. The lowest 40 bytes comprise the interrupt vector table (IVT). Each entry is four bytes and holds a relative branch instruction to an interrupt service routine (ISR). The remainder of the SRAM-backed memory range can be used for program and data. The highest 64 KB of the memory region is used for memory-mapped registers. A small portion of memory just above the IVT is reserved by the application binary interface (ABI).

0xFFFF	Special Core Registers
	Program counter, Status, Config
0xF0400	...
⋮	Reserved
0xF00FC	Memory-mapped general
	purpose registers.
0xF0000	R0-R63
⋮	Reserved for memory expansion
0x07FFF	Normal SRAM
	for program and data.
0x00028	Read/writable
0x00027	Interrupt Vector Table
	10 entries \times 4 bytes
0x00000	(branch instructions)

Figure 2.3: Epiphany core address map [9]

2.1.5 Memory model

The supported memory operations are *LOAD*, *STORE*, and *TESTSET* (atomic synchronization operation). All local memory accesses have a strong memory ordering, i.e., they take effect in the same order as they were issued. Memory accesses routed through one of the three NoCs have a weaker memory ordering. The router arbitration and dispatch rules are deterministic, but the programmer is not allowed to make assumptions regarding synchronization since there is no way to know the global “system state”.

Section 4.2 of the reference manual [9] lists the only guarantees on which the programmer can depend (summarized in table 2.1). It is important to note that in this context “previously written” means that the write (*STORE* operation) has propagated through the network, reached its destination router, and has been stored to its target address. Hence, if a write request is still *in-flight* (propagating through the network), a read request to the same address will return the old value if the read request reaches the destination router before the write request. Remember that read and write requests are routed through separate networks (*rMesh* and *cMesh*).

All local memory accesses have a strong memory ordering, i.e., they take effect in the same order as they were issued.

All memory requests that enter the NoC obey the following:

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

Table 2.1: Memory ordering guarantees [9]

2.2 The Parallella board

The Parallella board [8], depicted in fig. 2.4, is a credit-card sized microcomputer and is the de facto development board for the Epiphany architecture. It is based on a two-chip solution, a Xilinx Zynq 7000 family chip and an Epiphany-III chip. The Zynq system on a chip (SoC) is equipped with an ARM Cortex A9 processor and a field-programmable gate array (FPGA). The ARM processor is connected with the Epiphany chip's east eLink via an advanced extensible interface (AXI) bus bridge interface implemented in FPGA logic. This is shown in fig. 2.5.

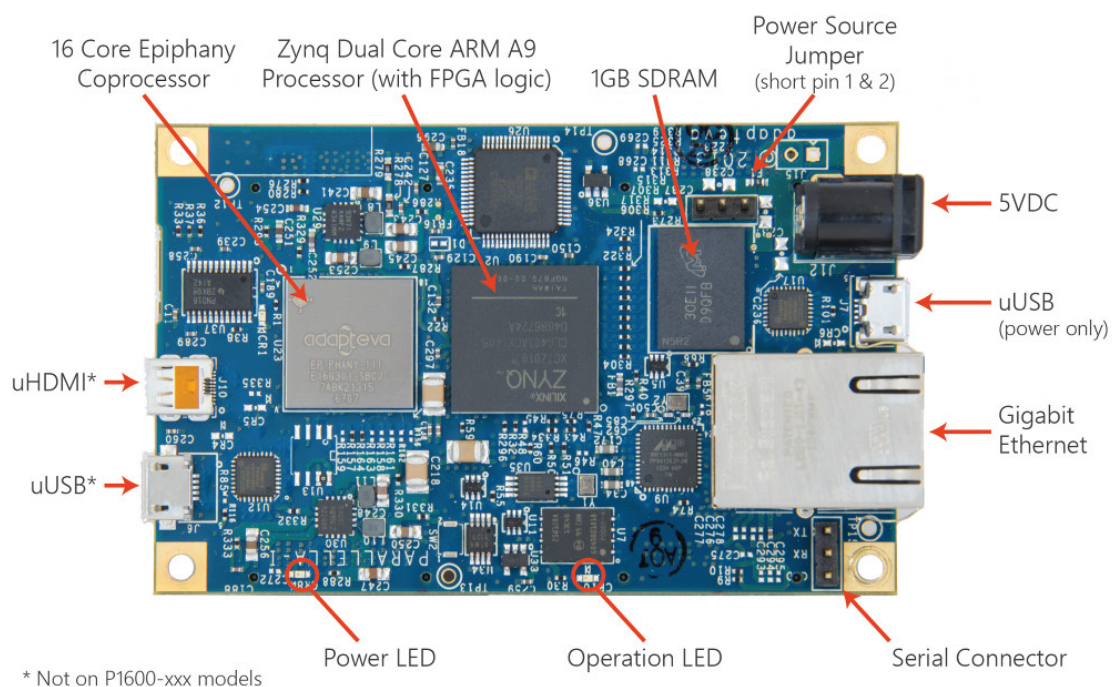


Figure 2.4: Parallella board

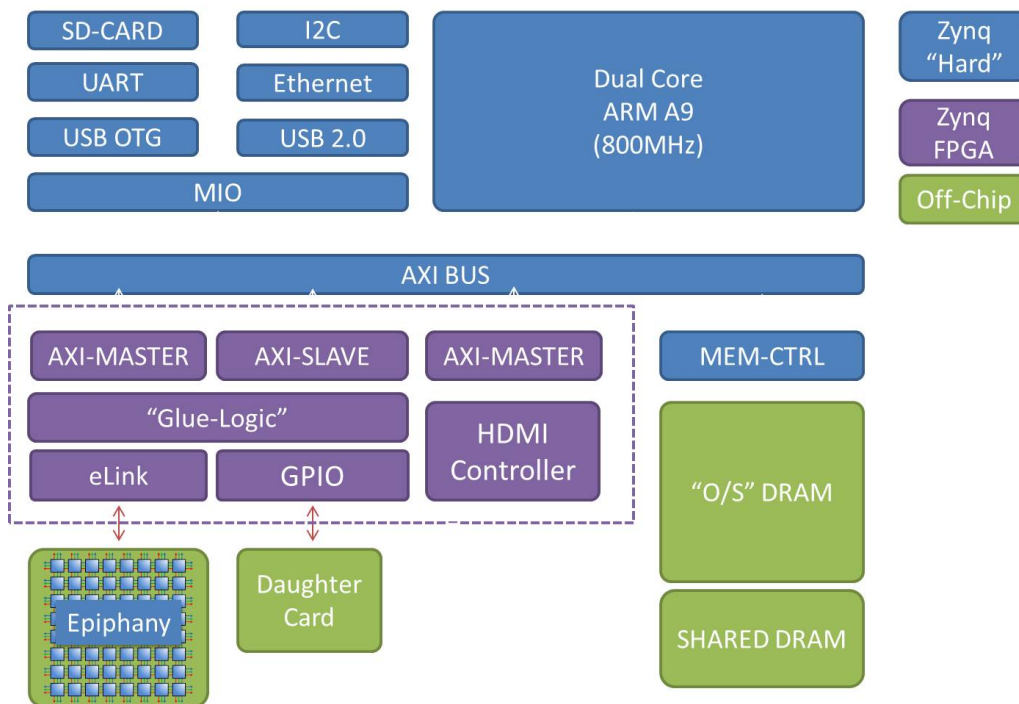


Figure 2.5: Parallella functional diagram [18]

2.3 The GNU debugger

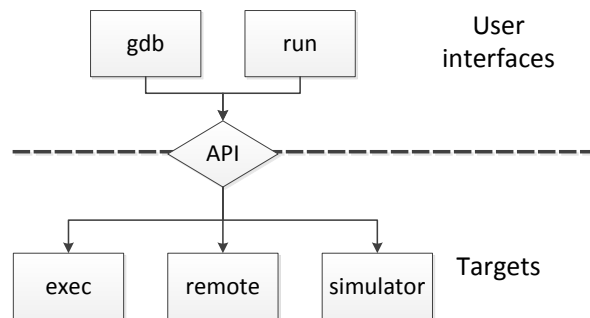


Figure 2.6: GDB overview

The simulator uses the common simulator framework for the GNU debugger (GDB), which is widely used and serves as the defacto standard debugger in the open-source community. Written by Richard Stallman in the 1980s, it was maintained by Cygnus Solutions throughout the 1990s until they merged with Red Hat in 1999. During this time GDB gained most of its target support, and many GDB-based simulators were written. Like the Adapteva simulator on which we base our work, most of these are for embedded systems.

GDB is divided into three main subsystems: user interface, target control interface, and executable file symbol handling [19]. See fig. 2.6 for an overview. Simulators are

mostly concerned with the user interface and target control interface. Compiling GDB with a simulator target creates two binaries, `epiphany-elf-gdb` and `epiphany-elf-run`. `epiphany-elf-gdb` is linked with the target simulator (in our case the Epiphany simulator) and presents the standard GDB user interface. `epiphany-elf-run` is a stand-alone tool that connects to the simulator target and runs a binary provided as a command-line argument.

2.3.1 Simulator framework

The GNU toolchain (compiler, assembler, linker, and tools) has been ported to many architectures over the years, and since writing a simulator in the process makes it easier to test generated code, GDB has acquired several simulator targets.

The process of adding a new architecture generally includes these steps:

- define the CPU components (register file, program counter, pipeline), instruction set binary format, and instruction semantics in a CPU definition file;
- write architecture-specific devices;
- write needed support code for a main loop generator script; and
- write simulator interface code.

The CPU definition file is written in an embedded Scheme-based, DSL. That definition is fed through CPU tools GENerator (CGEN) [20] to create C files for instruction decoding and execution within the simulator framework. Since code for the simulator interface and main loop tends to be similar across architectures, an existing simulator target can often be used as a base. For example, parts of the Epiphany implementation originate from the Mitsubishi M32R port. The CPU definition file is also used by other parts of the toolchain (`opcodes` and `as` (the GNU assembler)).

3

Implementation

This chapter describes in detail the simulator implementation and operation. The system can be seen as three main components:

- a single-core simulator,
- a eMesh NoC simulator, and
- a simulator frontend.

Figure 3.1 shows how we extend the single-core simulator to model a manycore system. Our design is process-based: for every core in the simulated system, we launch an `epiphany-elf-run` process. When the Epiphany simulator target is initialized, it also initializes the eMesh simulator, which connects to a shared memory file. The mesh network simulator uses POSIX shared memory to connect relevant portions of each core simulator via a unified address space, and all memory requests are routed through this eMesh simulator. The register file resides in the `cpu_state` structure. Since the eMesh simulator needs to access remote CPU state for some operations, we also store that state in the shared address space.

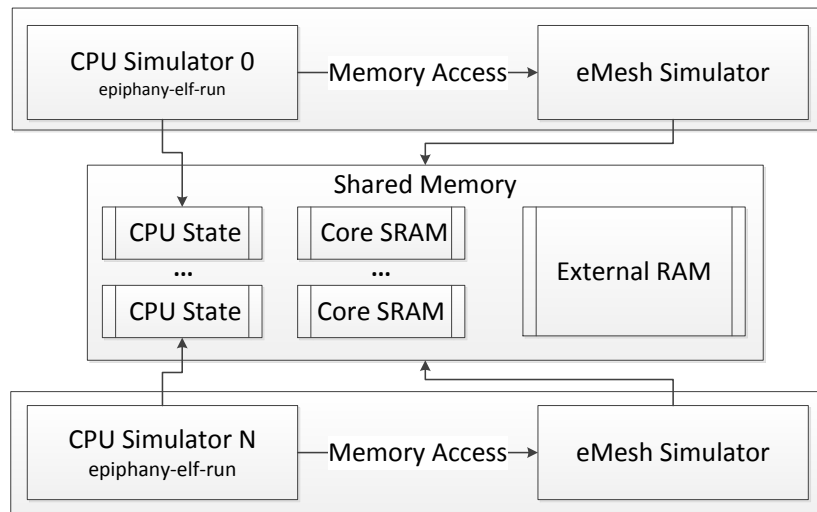


Figure 3.1: Simulator overview

3.1 Single-core simulator integration

We use the GDB Epiphany single-core simulator as a base in our work. Most instruction semantics had already been defined, and it was working with a single core. Due to the design of the GDB simulator framework, the simulator lacked support for self-modifying code in the modeled system. Due to the small local memories (32KB) in the Epiphany cores, executing code must be able to load instructions dynamically (like software overlays). Enabling self-modifying code required that we modify software mechanisms intended to speed simulation. For instance, a semantics cache maintains decoded target machine instructions, and in the original simulator code, writes to addresses in the semantics cache would update memory but not invalidate the instructions in the cache. The old code would still be executed. We added a flush mechanism.

We map the entire simulated 32-bit address space to a “shim” GDB device that forwards all memory requests to the eMesh network simulator. Recall that the CPU state of all cores resides in the shared address space, where the eMesh simulator can access it easily.

Algorithm 1 shows pseudocode for the simulator main loop. The highlighted lines are from the original single-core main loop. Lines 1-6 and 9 are inserted by the main loop generator script. The ISA has an *IDLE* instruction that puts the core in a low-power state and disables the program sequencer. We implement something similar in software: in line 8 we check whether the core is active, and if not, we sleep until we receive a wakeup event.

In line 13 we check whether another core has a pending write request to a special-core register (SCR). Writes to SCRs are serialized on the target core because they might alter internal core state. In lines 10 and 18 we handle out-of-band events. Such events might affect program flow and are triggered by writes to SCRs, e.g., by interrupts or reset signals. See section 3.1.1 for a description of the interrupt handling implementation.

In line 19 we ensure that only instructions inside the core’s local memory region can ever reside in the semantics cache. Without this constraint, we would need to do an invalidate call to all cores’ semantics caches on all writes. In line 21 we check whether the external write flag is set, and, if so, we flush the entire semantics cache. This flag is always set on a remote core when there is a write to that core’s memory.

3.1.1 Interrupt implementation

Figure 3.2 shows the state machine of the Epiphany interrupt controller. For the interrupt implementation, we use a simple event system which we incorporate into the main loop(algorithm 1). All writes to SCRs and instructions that might affect the interrupt state emit an interrupt event. In the main loop we then call `handle_out_of_band_events()`, which in turn calls `interrupt_handler()`, which implements the hardware side in fig. 3.2. The last piece of the puzzle was to implement the RTI (ReTurn from Interrupt) instruction.

Algorithm 1: Main loop (simplified for illustration)

Highlighted lines are the original main loop

```

1 while True do
2   sc  $\leftarrow$  scache.lookup(PC);
3   if sc =  $\emptyset$  then
4     insn  $\leftarrow$  fetch_from_memory(PC);
5     sc  $\leftarrow$  decode(insn);
6     scache.insert(PC, sc);
7   old_PC  $\leftarrow$  PC;
8   if core is in active state then
9     PC  $\leftarrow$  execute(sc);
10    PC  $\leftarrow$  handle_out_of_band_events(PC);
11  else
12    wait_for_wakeup_event();
13  if ext_scr_write_slot.reg  $\neq$  -1 then
14    reg_write(ext_scr_write_slot.reg,
15      ext_scr_write_slot.value);
16    ext_scr_write_slot.reg  $\leftarrow$  -1;
17    signal_scr_write_slot_empty();
18    PC  $\leftarrow$  handle_out_of_band_events(PC);
19  if old_PC  $\notin$  local memory region then
20    scache.invalidate(old_PC);
21  if external_mem_write_flag then
22    scache.flush();
23    external_mem_write_flag  $\leftarrow$  False;

```

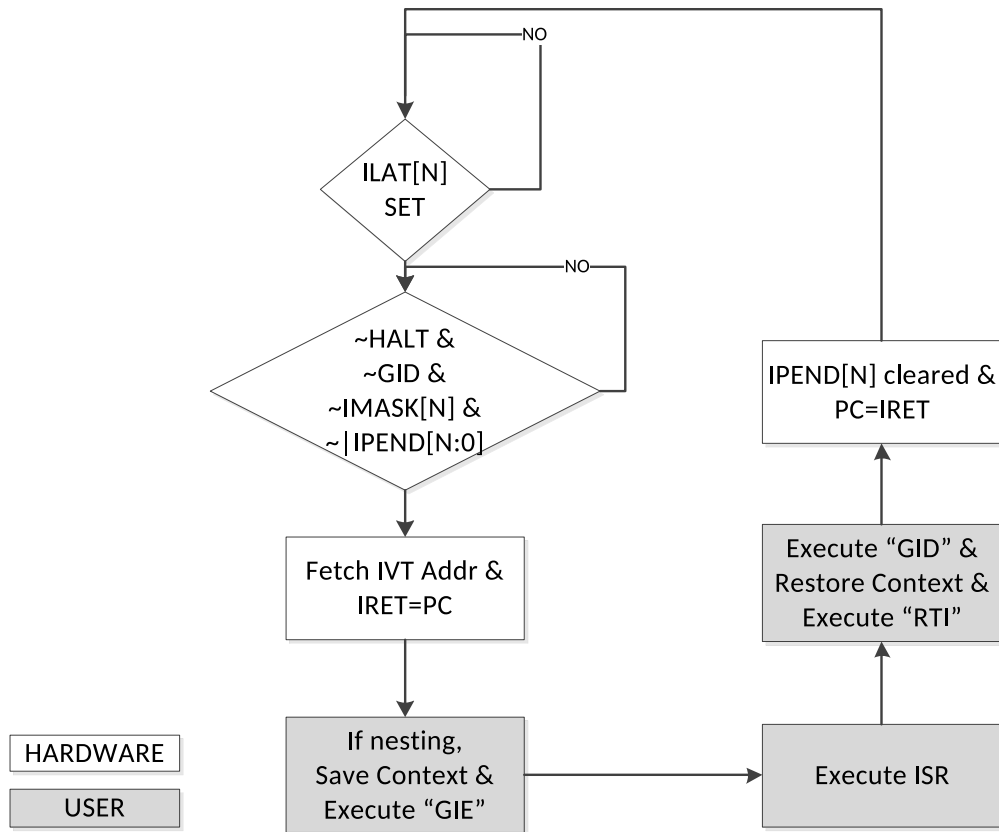


Figure 3.2: Interrupt service routine operation [9]

3.1.2 Hardware loops

Hardware loops are controlled by the LC (loop counter), LS (loop start address), and LE (loop end address) registers. When the next PC (program counter) is equal to LE, and LC is non-zero, the program will jump to LS, and LC will be decremented by one. We implement hardware loops by special-casing the handling of the PC in the CGEN CPU definition file.

3.2 Device implementation

3.2.1 Timers

Our timer implementation supports the *CLK* and *IDLE* tick event types, which are the most commonly used event types and a requirement of some of the *epiphany-examples* programs. It also supports a *CHAINED* mode that combines the two available timers (*timer0* and *timer1*) into a 64-bit wide counter. This feature is only available on Epiphany-IV and later chips.

3.2.2 DMA

Our DMA implementation supports everything except slave-mode DMA. Slave-mode DMA is a *LABS* feature present in Epiphany-III and Epiphany-IV, but which is likely to be removed in later generations of the Epiphany processor.

3.3 eMesh simulator

As shown in fig. 3.1, the eMesh simulator creates a shared address space accessible to all simulated cores. This is accomplished via the POSIX shared memory application programming interface (API). We use POSIX threads (pthreads) for inter-process communication.

The eMesh simulator provides an API for the (*LOAD*, *STORE*, and *TESTSET*) memory transactions, along with functions to connect and disconnect to the shared address space. We also provide a client API so that other applications can access the Epiphany address space (e.g., to model the external host or instrument a simulated application).

Every memory request must be translated. The translator maps an Epiphany address to its corresponding location in the simulator address space. It also determines to which type of memory (core SRAM, external DRAM, memory-mapped registers, or invalid) the address corresponds.

How the request is serviced depends on the memory type. Accesses to core SRAM and external DRAM are implemented as native load and store operations (the target core need not be invoked). Memory-mapped registers are a little trickier. All writes to such registers are serialized on the target core. This is accomplished with one write slot, a mutex, and a condition variable. Reads from memory-mapped registers are implemented as normal load operations.

Since reads to memory-mapped, general-purpose registers are only allowed when the target core is inactive, we check core status before allowing the request.

3.4 Parallella/Epiphany SDK support

We created a backend for the Epiphany hardware abstraction library (e-hal). By using the eMesh simulator client API, we can compile Parallella host applications natively for x86_64 without code modification¹ This is shown in fig. 3.3. We experimented with cross-compiling programs (from the `epiphany-examples` repository on the Adapteva GitHub account) with generally good results. Obviously, programs that use implicit synchronization or depend on hardware timings might not work, and programs that use core functionalities not yet supported will not work.

¹Since data structures are passed between the host (64-bit x86_64), and target (32-bit Epiphany), their memory layout must be explicitly defined.

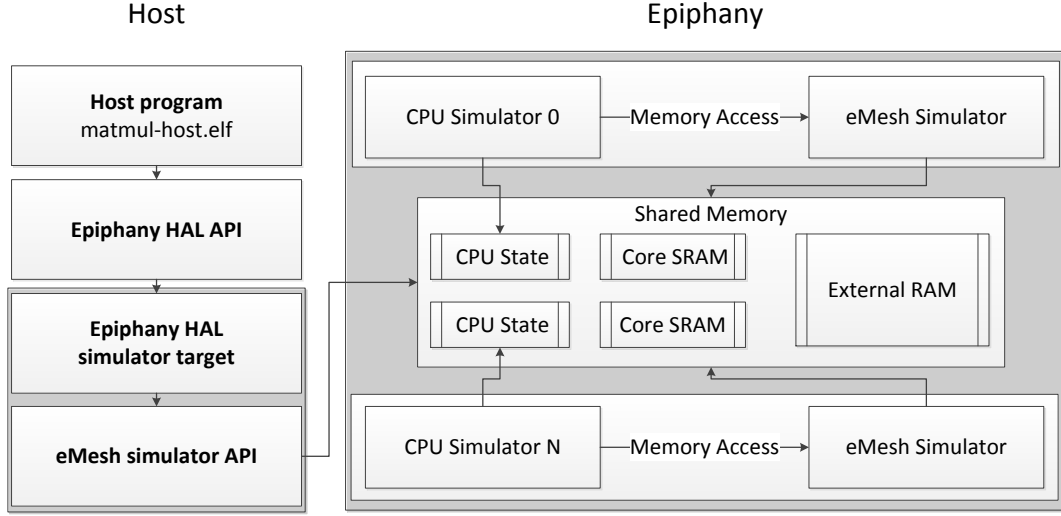


Figure 3.3: Host device communication

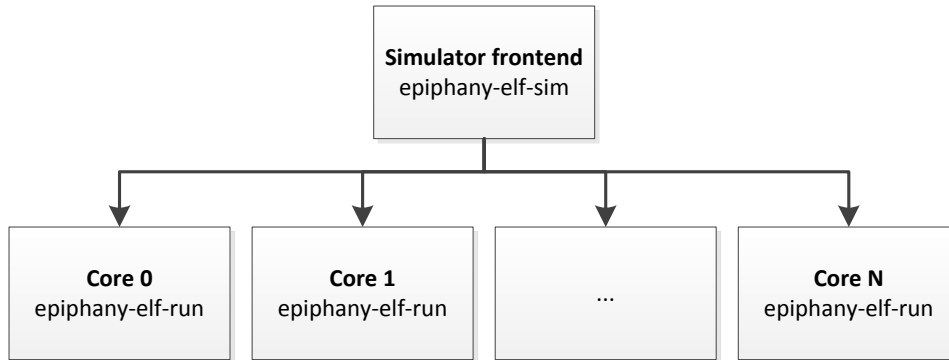
3.5 Networking support

We have also extended the eMesh simulator with networking support implemented in MPI [21]. We use MPI’s remote memory access (RMA) API to implement normal memory accesses (core SRAM and external RAM). We implement all register accesses with normal message passing and a helper thread on the remote side. We implement *TESTSET* with `MPI_compare_and_swap()`, which is only available in MPI-3.0 [22]. Since we use both threads and MPI-3.0 functionalities, we require a fairly recent MPI implementation compiled with `MPI_THREADS_MULTIPLE` support.

We divide the simulation into sub-tiles, which we map to the network nodes allocated for the simulation. The eMesh simulator uses the single-host shared memory implementation for accessing simulated cores that reside on the same node. MPI will only be invoked when the target core simulator resides on a different node.

3.6 Simulator frontend

As noted, the simulator is process-based, i.e., one simulated core maps to one system process. When we began development, we started these processes by hand (which is cumbersome and does not scale). We therefore created a command-line tool called *epiphany-elf-sim*, which makes it easy to launch simulations. Mesh properties and the program(s) that should be loaded onto the cores are given as command-line arguments, and the tool spawns and manages the core simulator processes. Full program usage is listed in Appendix A.

**Figure 3.4:** Simulator frontend overview**Listing 3.1:** Example 8x8 epiphany-elf-sim invocation

```
$ epiphany-elf-sim -r 8 -c 8 --redirect-dir out ./hello-world.elf
```

Listing 3.2: Example epiphany-elf-sim invocation with a host program

```
$ epiphany-elf-sim --redirect-dir out --host ./matmul-host.elf
```

3.7 Simulation environments

The GDB simulator framework provides three different simulation environments, user, virtual, and operating. User environment is intended to simulate normal user programs whereas operating environment is for OSs or bare-metal type applications. It is not clear what the virtual environment should be used for: different ports implement it slightly differently. This is how we map the environments: `user` is the default for running single-threaded applications with `epiphany-elf-run`; `virtual` is the default for starting a stand-alone simulation (without a host) with `epiphany-elf-sim`; and `operating` is the default when `epiphany-elf-sim` is started in host-device mode (the `--host` argument). Interrupts are not supported in user environment, and any interrupt or exception will cause the simulation to halt. Also, any instruction that would put the core in an inactive state (e.g., the `IDLE` instruction) will stop the simulation. Interrupts are supported in the virtual and operating environments, and a core can become inactive without the simulation stopping. The only difference between how we implement the virtual and operating environments is how a core is started. In the virtual environment we trigger a sync interrupt, and in the operating environment the core starts up in the inactive state, just as on the chip. It is then the responsibility of a host program to load a binary image onto the core and start it by sending a sync interrupt. We let the simulator handle traps in all three environments.

4

Results

The simulator supports all features not marked as *LABS* in the reference manual (i.e., untested or broken functionalities), with the exception that the event timers only support CLK and IDLE as event sources. The simulator executes millions of instructions per second (per physical host core) and scales up to 4095 simulated cores running concurrently on a single computer. For the networking backend, we have run tests with up to 1024 simulated cores spread over up to 48 nodes in an HPC environment. In larger single-node simulations the memory footprint averages under 6MB per simulated core. Note that the simulator design requires that writes from non-local (external) cores flush the entire semantics cache (see algorithm 1, line 21–23) rather than just invalidating the affected region, which may hamper performance.

4.1 Epiphany examples

The `epiphany-examples` [23] repository contains a variety of self-contained example programs that demonstrate how the Epiphany architecture and software libraries can be used for solving basic tasks. These include (i) common math problems (such as parallel FFT and matrix-multiplication), (ii) code more specific to the architecture (such as DMA, timers, and interrupts), and (iii) pure library and platform regression tests. Since all the examples use the Epiphany libraries and the simulator is a supported e-hal backend, the `epiphany-examples` test suite can be run against the simulator in place of real hardware. This is demonstrated in fig. 4.1. As can be seen in the figure, the test suite passes when run against the simulator. A few test cases are expected to fail (indicated by *CROSS_XFAIL*). These failures are well understood and are caused by (i) timing assumptions explicit to the Epiphany-III chip or Parallella board (`cpu/mutex`, `dma/dma_message_read`), (ii) DMA slave mode not supported by the simulator (`dma/dma_slave`), and (iii) an addressing regression test for the Parallella FPGA eLink/AXI bridge driver that is not applicable to the simulator (`test/elink-rx-remapping`).

4. Results

```
~/.../epiphany-examples$ ./scripts/build_and_test_all.sh
Detected non-Parallellla host
Instructing build scripts to NOT cross-compile host programs
Using simulator for tests
```

Phase 1: Build and run all tests once

Directory	Build	Test
apps/dotproduct	OK	EXIT_OK
apps/e-bandwidth-test	OK	EXIT_OK
apps/e-dump-mem	OK	EXIT_OK
apps/e-dump-regs	OK	SKIP
apps/e-fill-mem	OK	EXIT_OK
apps/e-mem-sync	OK	EXIT_OK
apps/eprime	OK	SKIP
apps/erm	OK	SKIP
apps/erm_example	OK	SKIP
apps/e-toggle-led	OK	N/A
apps/fft2d	OK	EXIT_OK
apps/hello-world	OK	EXIT_OK
apps/matmul-16	OK	EXIT_OK
apps/matmul-64	OK	SKIP
apps/shm_test	OK	EXIT_OK
archive/e-init	SKIP	SKIP
archive/e-standby-test	SKIP	SKIP
archive/e-test	SKIP	SKIP
cpu/arithmode	OK	EXIT_OK
cpu/assembly	OK	EXIT_OK
cpu/basic_math	OK	EXIT_OK
cpu/complex_numbers	OK	EXIT_OK
cpu/ctimer	OK	SKIP
cpu/interrupt-demo	OK	EXIT_OK
cpu/interrupts	OK	EXIT_OK
cpu/mutex	OK	CROSS_XFAIL
cpu/nested_interrupts	OK	EXIT_OK
cpu/register_test	OK	EXIT_OK
cpu/remote_call	OK	EXIT_OK
dma/dma_2d	OK	EXIT_OK
dma/dma_chain	OK	EXIT_OK
dma/dma_interrupt	OK	EXIT_OK
dma/dma_message_read	OK	CROSS_XFAIL
dma/dma_message_write	OK	EXIT_OK
dma/dma_slave	OK	CROSS_XFAIL
emesh/emesh_bandwidth_all2one	OK	EXIT_OK
emesh/emesh_bandwidth_bisection	OK	EXIT_OK
emesh/emesh_bandwidth_neighbour	OK	EXIT_OK
emesh/emesh_read_latency	OK	EXIT_OK
emesh/emesh_traffic	OK	EXIT_OK
errata/noc_fifo	OK	EXIT_OK
io/link_lowpower_mode	OK	SKIP
labs/clockgating_mode	OK	SKIP
labs/hardware_barrier	OK	SKIP
labs/hardware_loops	OK	SKIP
labs/mailbox-test	OK	SKIP
labs/mem_protect	OK	SKIP
softcache/basic_math	OK	EXIT_OK
softcache/math_example	OK	EXIT_OK
test/e-extmem-test	OK	N/A
test/e-free-resource-leak-test	OK	EXIT_OK
test/e-loopback-test	OK	SKIP
test/e-matmul-test	SKIP	SKIP
test/e-mem-test	OK	EXIT_OK
test/e-mesh-test	OK	EXIT_OK
test/e-read-buf	OK	EXIT_OK
test/e-read-word	OK	EXIT_OK
test/e-regfile-test	OK	EXIT_OK
test/e-reset	OK	EXIT_OK
test/e-test	SKIP	SKIP
test/e-write-buf	OK	EXIT_OK
test/e-write-word	OK	EXIT_OK
test/test-elink-rx-remapping	OK	CROSS_XFAIL
test/test-linker-script-symbols	OK	EXIT_OK
test/test-load-elf-w-empty-segment	OK	EXIT_OK

Cross compilation detected: Skipping repeated tests.

STATUS: PASS

Figure 4.1: epiphany-examples test suite

4.2 Domino

The Domino demo shows that a 4095 core chip can be simulated on one machine. The aggregated memory usage was less than 6MB per simulated core. The example tests inter-core communication, “model scalability”, and the interrupt implementation.

1. All cores except first calls “IDLE”
2. Leader:
 - (a) sends its coreid to next core
 - (b) triggers message interrupt on next
 - (c) calls “IDLE”
3. Next
 - (a) appends its coreid and sends to next
 - (b) triggers message interrupt on next
 - (c) calls exit
4. ...
5. Last core sends message to leader
6. Leader traverses list and prints out the route the message took

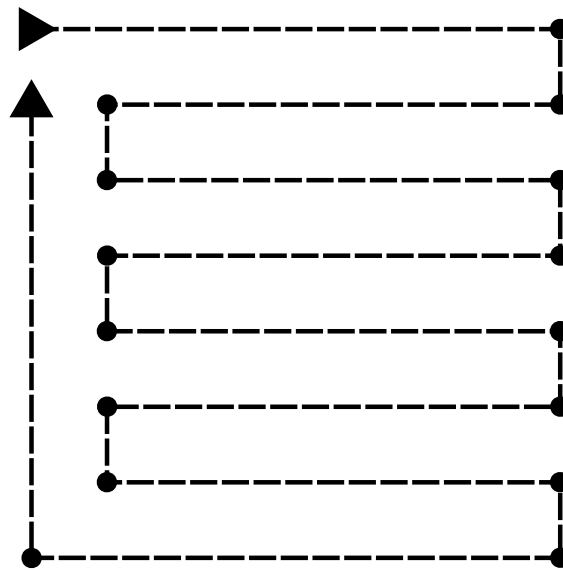


Figure 4.2: Domino message path

4.3 HPC demonstration

The parallel matrix multiplication application from `epiphany-examples` uses many CPU functions and performs all interesting work on the Epiphany chip, and thus, we choose it for initial studies of simulator behavior. For simplicity, we move all host code to the first core simulator process. For this test, we implemented our own data transfer functions since the simulator did not yet support DMA at the time. Our port revealed a race condition in the `e-lib` barrier implementation, which we attempted to fix (see the discussion of the 1024-core simulation, below).

For the tests, we allocated 32 nodes on an HPC cluster. Nodes contain two Intel Xeon processors with 16 physical cores (eight per socket) connected by a Mellanox Infiniband FDR. Hyper-threading is disabled. Tables 4.2-4.5 present results for 16, 64, 256, and 1024 simulated cores per computation, respectively.

Things to consider:

1. Using more nodes creates more network traffic (versus direct accesses to memory within a node) for handling simulated memory accesses. This is orders of magnitude slower, even with FDR Infiniband.
2. More nodes means more physical cores. If all simulated cores only accessed their local memory, the ideal number of nodes would be where there was a one-to-one mapping between physical cores and simulated cores.
3. The `e-lib` barrier implementation does busy-waiting, and thus waiting for another core increases total instructions executed. This also shows a clear limitation of the simulator. Because there is no global time or rate-limiting, the number of executed instructions could differ significantly between the simulator and real hardware.
4. The nodes are allocated with exclusive access, but the network is shared between all users of the cluster. This means that there might be network contention or other system noise that could qualitatively affect results. Another error factor is the network placement of the nodes for a requested allocation, which is also beyond our control.

For `matmul-16` and `matmul-64`, results are understandable: the rate of instructions executed per simulated core increases until we reach a 2:1 mapping between physical and simulated cores. The reason that 2:1 outperforms 1:1 is likely because all simulator processes running on a node can be pinned to physical cores on the same socket, which means more shared caches and less cache coherence communication between sockets.

Execution time increases for a modest number of simulated cores when we go from one to two nodes due to network communication. We get a nice speedup in execution time for `matmul-64`.

For `matmul-256` two results stand out. The jump in execution time from one to two nodes is much higher compared to `matmul-16` and `matmul-64`. This data point might be due to a system/network anomaly: unfortunately, we only ran this test once (to date), so we cannot yet explain the observed behavior.¹ For `matmul-256` running on 32 nodes, execution time jumps from 44 seconds on 16 nodes to 521 seconds on 32 nodes. We could expect execution time to increase a bit, since there is a 1:1 mapping between physical and simulated cores on 16 nodes, and we get more network traffic with 32 nodes. We repeated the test a few times (on the same allocation) with similar results. This behavior, too, requires further study.

When we tried to scale up to 1024 simulated cores, the program could not run to completion. Attaching the debugger revealed that all simulated cores were stuck waiting on the same barrier. It is likely that we hit the `e-lib` race condition and that our fix proved insufficient. A proper fix has since been applied for the upstream `e-lib` barrier implementation².

As a limit study, we removed all barrier synchronization. This means that program

¹In truth, our allocation of CPU hours expired before we could repeat all our experiments.

²<https://github.com/adapteva/epiphany-libs/commit/67b6b63f>

output is incorrect, and the results for matmul-1024 are not directly comparable to the other runs. Since all synchronization is removed, we expect execution time to be lower than it would have been in a correct implementation. However, the program still exhibits a similar memory access pattern, so it is fair to assume that the instruction rate tells something about performance even with synchronization back in place. From the previous results, we would expect running on 128 nodes to yield the lowest execution time and peak instruction rate. Running on a larger allocation is part of future work.

Legend	Description
N	The number of nodes used in the simulation.
Δt	The execution time of the simulation.
Σ_{insns}	The aggregate number of executed instructions for all simulated cores.
\min_{insns} \max_{insns}	The number of executed instructions for the simulated core with the minimum and maximum executed instructions, respectively.
$\langle \text{insns/core/s} \rangle$	The average rate of executed instructions per core in the simulation.
$\langle \text{insns/s} \rangle$	The aggregate rate of executed instructions for all simulated cores.

Table 4.1: Legend description for tables 4.2 to 4.5

N	Δt	Σ_{insns}	\min_{insns}	\max_{insns}	$\langle \text{insns/core/s} \rangle$	$\langle \text{insns/s} \rangle$
1	32.0 s	2.28E+09	1.40E+08	1.44E+08	4.45E+06	7.12E+07
2	37.9 s	4.22E+09	2.07E+08	3.29E+08	6.96E+06	1.11E+08
4	53.2 s	5.13E+09	2.49E+08	4.68E+08	6.03E+06	9.65E+07
8	65.8 s	5.51E+09	2.42E+08	5.55E+08	5.24E+06	8.38E+07

Table 4.2: Results for matmul-16

N	Δt	Σ_{insns}	\min_{insns}	\max_{insns}	$\langle \text{insns/core/s} \rangle$	$\langle \text{insns/s} \rangle$
1	81.6 s	6.01E+09	9.22E+07	9.52E+07	1.15E+06	7.37E+07
2	88.6 s	1.30E+10	1.26E+08	2.42E+08	2.30E+06	1.47E+08
4	46.5 s	1.29E+10	1.30E+08	2.52E+08	4.34E+06	2.78E+08
8	29.2 s	1.42E+10	1.33E+08	3.05E+08	7.60E+06	4.86E+08
16	29.9 s	1.46E+10	1.29E+08	3.16E+08	7.63E+06	4.88E+08
32	35.1 s	1.65E+10	1.52E+08	3.70E+08	7.35E+06	4.71E+08

Table 4.3: Results for matmul-64

N	Δt	Σ_{insns}	\min_{insns}	\max_{insns}	$\langle \text{insns/core/s} \rangle$	$\langle \text{insns/s} \rangle$
1	344.3 s	2.10E+10	7.45E+07	8.51E+07	2.38E+05	6.10E+07
2	1007.3 s	1.52E+11	1.13E+08	9.18E+08	5.89E+05	1.51E+08
4	323.8 s	1.08E+11	1.51E+08	5.08E+08	1.31E+06	3.35E+08
8	103.0 s	6.98E+10	9.17E+07	3.53E+08	2.65E+06	6.77E+08
16	44.4 s	6.03E+10	1.36E+08	2.49E+08	5.31E+06	1.36E+09
32	520.9 s	1.58E+12	8.39E+07	6.32E+09	1.19E+07	3.04E+09

Table 4.4: Results for matmul-256

N	Δt	Σ_{insns}	\min_{insns}	\max_{insns}	$\langle \text{insns/core/s} \rangle$	$\langle \text{insns/s} \rangle$
16	262.8 s	1.50E+11	3.12E+07	1.47E+08	5.59E+05	5.73E+08
32	135.5 s	1.50E+11	3.12E+07	1.47E+08	1.08E+06	1.11E+09

Table 4.5: Results for matmul-1024*

4.4 Simulator frontend

The simulator frontend (`epiphany-elf-sim`) makes it easy to start simulations. You specify the mesh layout, external memory base and size, and which binary(ies) to load. It starts and manages the core simulator processes. It supports redirecting standard input and output to per-core files. It defaults to the Parallella configuration if no options are provided.

4.5 Integration with Epiphany SDK

e-hal supports using the simulator as a target. The e-hal simulator target can be enabled by providing the `--host` flag to `epiphany-elf-sim`. This makes it easy to run programs written for the SDK against the simulator, without any code modification.

5

Discussion

The simulator was written with the goals of functional correctness, scalability, and performance (i.e., instructions executed per time unit). In the previous chapter, we have shown that our implementation achieves these goals.

- In section 4.1, we demonstrate that the simulator is functionally correct.
- In section 4.2, we show that the simulator can model systems with up to 4095 cores.
- In section 4.3, we show that a simulation can be distributed across multiple network nodes. We present results with up to 1024 simulated cores spread over 32 compute nodes.

We have shown that the simulator is comparably fast, functionally correct, and that our distributed networked implementation is a viable approach for scaling a manycore simulator beyond thousands of simulated cores while still reaping the benefits of the added parallelism.

The good simulation speed is achieved due to mainly two factors. We piggy-back on the memory ordering of the host system on which the simulator is running. Also, we have yet to implement a global timing model for the simulated cores. We describe how the latter can be implemented in section 5.1.1.

Piggy-backing on the memory consistency model of the host system demands that the host has an equal or stronger memory ordering guarantee than the simulated architecture. Otherwise, the simulator cannot be functionally correct with this approach. There is one more caveat: if the host system has *stronger* memory ordering (which is the case for x86_64 and ARM64 vs. Epiphany), that means that the simulator is still functionally correct. I.e., an application adhering to the memory model of the simulated architecture will have a correct execution on the simulator. But a host system with a stronger memory ordering guarantee can also hide memory ordering bugs in a user application. The application will run fine on the simulator but break on real hardware.

The best use case for the simulator in its current state is for developers who wish to design, test, and debug their applications. Debugging capabilities exist but are limited. In section 5.1.2, we describe how the debugging experience can be improved.

Researchers and computer architects will have a harder time using the simulator due to its lack of a decent timing model. Their use case requires more accurate timing information to be able to evaluate if changing a design parameter, extending the ISA, or adding hardware is worthwhile. We want to emphasize that the timing model does not need to be perfectly cycle-accurate; gem5 [13] is not cycle-accurate, but its timing is good enough to be widely used in this field.

We believe that a good timing model is an important component in a computer system simulator, and that our simulator's lack of such a model limits its usability. Further,

the timing granularity should ideally be a tunable knob. Different use cases require different levels of timing accuracy, and more fine-grained timing will decrease simulation speed due to the need for more synchronization.

5.1 Future work

5.1.1 Clock count estimation

The simulator lacks a good model for the number of clock cycles a program would take to run on real hardware. The simulator can give a detailed summary of executed instructions and memory accesses, but that is it.

Three components are needed for better timing analysis: a CPU pipeline model, a network delay model, and a global clock. The GDB simulator framework supports CPU pipeline modeling, so that should be reasonably easy to implement. With a pipeline model, the simulator would report accurate timing results for single-threaded programs.

In fig. 5.1 we show a simple formula for modeling the network delay. C is the number of stall cycles, and $M(a, b)$ is the Manhattan distance between core a and b . The 8 factor is the number of clock cycles per router hop on the *rMesh* network, and 1.5 is the number of clock cycles per router-hop on the *cMesh* (the return path). The model is applicable for *LOAD* and *TESTSET* mesh transactions. *STORE* transactions incur no stall cycles. The model assumes no network congestion.

$$C = (8 + 1.5) \times M(a, b)$$

Figure 5.1: NoC delay formula

The last component needed for a better timing model is a global clock. Since the simulator uses one system process per simulated core, it is the task of the OS to schedule the simulator processes fairly. The OS's concept of fairness is equal execution time. Simulated cores might execute different instructions. Different instructions take different amounts of time to simulate. This makes the clock cycle count drift among simulated cores over time. As long as there is no communication among threads, this is not an issue, threads will have different simulation times, but the clock count will stay correct. However, once synchronization (e.g., mutexes and barriers) is added into the mix, threads become interdependent, and the clock drift spills over to different threads, and *that* skews the clock count. This is even worse if the simulation processes are spread over more than one node across a network.

The solution is to add a global clock to the simulation. In the most extreme case, the simulator processes would have a synchronization point across all cores for *every* simulated clock cycle. This will likely be unacceptably slow, especially over a network. Instead, we suggest adding a barrier every k simulated clock cycle. A smaller k means better clock accuracy at the price of worse performance. The parameter can be tuned to suit the user's needs.

These measures will not make the simulator cycle-accurate, but we believe that it will provide an estimate that is good enough to meet the requirements of application developers and architecture researchers.

5.1.2 Multicore debugging support

Debugging multi-threaded applications can be a real challenge. Epiphany programs can be debugged via the remote GDB protocol. Adapteva ships its own remote GDB server, which is called `e-server`¹. `e-server` has recently gained support for *non-stop* and *multiprocess*, two advanced GDB debugging features [24, 25]. This allows for debugging multi-threaded programs efficiently on the Parallella board. It would also be beneficial if a simulator target were implemented for `e-server`, similarly to how `e-hal` has a simulator target.

¹<https://github.com/adapteva/epiphany-libs/tree/master/e-server>

6

Conclusion

In this thesis, we describe the implementation of a manycore simulator for the Adapteva Epiphany architecture. The work is based on an existing single-core simulator. The single-core simulator has been extended and integrated with our mesh NoC simulator. This has enabled us to do full-chip simulations modeling large numbers of cores. We have shown that the simulator is functionally correct, scalable, and has good performance.

The work in this thesis was included in the official Epiphany SDK in 2016. Since then the SDK has been downloaded over 18,000 times. It is fair to say that the simulator has reached a wide audience, even though we do not know how many people have actually used it. The Adapteva Parallella community has over 7,000 registered members, and over 10,000 evaluation boards have been sold to date. All source code is available from <https://github.com/adapteva/epiphany-binutils-gdb.git>. We welcome others who would like to contribute to the simulator's further development.

Bibliography

- [1] O. Jeppsson and S. A. McKee. “Towards a scalable functional simulator for the Adapteva Epiphany architecture”. In: *Proceedings of 8th Annual Workshop on Programmable Issues for Heterogeneous Multicores (MULTIPROG)*. 2015. URL: <http://research.ac.upc.edu/multiprog/multiprog2015/papers/multiprog-2015-13.pdf>.
- [2] F. Pollack. *Pollack’s Rule of Thumb for Microprocessor Performance and Area*. 2007. URL: https://en.wikipedia.org/wiki/Pollack's_Rule (visited on 06/2019).
- [3] B. de Dinechin et al. “A clustered manycore processor architecture for embedded and accelerated applications”. In: *Proceedings of 17th High Performance Extreme Computing Conference (HPEC)*. IEEE Computer Society, 2013. DOI: 10.1109/HPEC.2013.6670342.
- [4] A. M. Jones and M. Butts. “TeraOPS hardware: A new massively-parallel MIMD computing fabric IC”. In: *IEEE 18th Hot Chips Symposium (HC18)*. 2006. DOI: 10.1109/HOTCHIPS.2006.7477853.
- [5] B. Bohnenstiehl et al. “KiloCore: A 32-nm 1000-Processor Computational Array”. In: *IEEE Journal of Solid-State Circuits* 52.4 (2017), pp. 891–902. DOI: 10.1109/JSSC.2016.2638459.
- [6] A. Olofsson. “Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip”. In: *CoRR* abs/1610.01832 (2016). arXiv: 1610.01832.
- [7] A. Olofsson, T. Nordström, and Z. Ul-Abdin. “Kickstarting high-performance energy-efficient manycore architectures with Epiphany”. In: *Proceedings of 48th Asilomar Conference on Signals, Systems and Computers*. IEEE, 2014, pp. 1719–1726. DOI: 10.1109/ACSSC.2014.7094761.
- [8] *Parallella board website*. URL: <https://www.parallella.org> (visited on 06/2019).
- [9] Adapteva Inc. *Epiphany Architecture Reference*. 2014. URL: http://www.adapteva.com/docs/epiphany_arch_ref.pdf.
- [10] F. Bellard. “QEMU, a Fast and Portable Dynamic Translator”. In: *Proceedings of USENIX Annual Technical Conference*. USENIX Association, 2005, pp. 41–46.
- [11] “Multi-threaded emulation for QEMU”. In: *lwn.net* (2015). URL: <https://lwn.net/Articles/697265/> (visited on 06/2019).

- [12] J.-H. Ding et al. "PQEMU: A parallel system emulator based on QEMU". In: *Proceedings of 17th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2011, pp. 276–283.
- [13] N. Binkert et al. "The Gem5 Simulator". In: *SIGARCH Computer Architecture News* 39.2 (2011). ACM. doi: 10.1145/2024716.2024718.
- [14] *SID Simulator homepage*. URL: <https://www.sourceware.org/sid/> (visited on 06/2019).
- [15] R. Lantz. "Fast functional simulation with parallel Embra". In: *Proceedings of 4th Annual Workshop on Modeling, Benchmarking and Simulation (MOBS)*. 2008.
- [16] J. E. Miller et al. "Graphite: A distributed parallel simulator for multicores". In: *Proceedings of 16th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2010.
- [17] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Standard 754-2008* (2008). doi: 10.1109/IEEESTD.2008.4610935.
- [18] Adapteva Inc. *Parallella-1.x Reference Manual*. 2016. URL: https://www.parallella.org/docs/parallella_manual.pdf.
- [19] J. Gilmore and S. Shebs. *GDB Internals*. Tech. rep. 1991–2013.
- [20] D. Evans and et al. *CGEN: CPU tools GENerator*. URL: <https://sourceware.org/cgen/>.
- [21] D. W. Walker. "The design of a standard message passing interface for distributed memory concurrent computers". In: *Parallel Computing* 20.4 (1994). Elsevier, pp. 657–673. doi: 10.1016/0167-8191(94)90033-7.
- [22] M. P. I. Forum. *MPI: A Message-Passing Interface Standard Version 3.0*. Tech. rep. 2012.
- [23] *epiphany-examples github Repository*. URL: <https://github.com/adapteva/epiphany-examples>.
- [24] N. Sidwell et al. "Non-stop Multi-Threaded Debugging in GDB". In: *Proceedings of GCC Developers' Summit*. 2008, pp. 117–128.
- [25] P. Alves. "GDB, so where are we now? - Status of GDB's ongoing target and run control projects". In: *Slides from FOSDEM 2014 conference*. 2014. URL: https://archive.fosdem.org/2014/schedule/event/gdb_target_run_valgrind/attachments/slides/393/export/events/attachments/gdb_target_run_valgrind/slides/393/pedro_alves_gdb_slides.pdf.

Acronyms

ABI	application binary interface. 7
ALU	arithmetic logic unit. 5
API	application programming interface. 15, 16
AXI	advanced extensible interface. 8, 19
CGEN	CPU tools GENerator. 3, 10, 14
DMA	direct memory access. 5, 21
DSL	domain-specific language. 3, 10
e-hal	Epiphany hardware abstraction library. 15, 19, 24
FPGA	field-programmable gate array. 8
FPU	floating-point unit. 5
GDB	the GNU debugger. xi, 3, 5, 9, 10, 12, 17, 26, 27
HPC	high-performance computing. 2, 3, 19, 22
IC	integrated circuit. 6
ILP	instruction-level parallelism. 1
ISA	instruction set architecture. 2, 3, 5, 12, 25
ISR	interrupt service routine. 7
IVT	interrupt vector table. 7
MIMD	multiple instruction streams, multiple data streams. 1
MMU	memory management unit. 5
MPI	message passing interface. v, 2, 3, 16
NoC	network-on-chip. 1, 2, 5–7, 11, 29
OS	operating system. 2, 3, 17, 26
RISC	reduced instruction set computing. 1, 5
RMA	remote memory access. 16
RTL	register-transfer level. 1

SCR	special-core register. 12
SDK	software development kit. v, ix, 2, 15, 24, 29
SoC	system on a chip. 8

Appendices

A

Simulator frontend usage

```
usage: epiphany-elf-sim [-h] [--verbose] [-r ROWS] [-c COLS] [-f FIRST_CORE]
                        [-i FIRST_ROW] [-j FIRST_COL]
                        [--environment ENVIRONMENT]
                        [--ext-ram-size EXT_RAM_SIZE]
                        [--ext-ram-base EXT_RAM_BASE]
                        [--redirect-dir REDIRECT_DIR]
                        [--wait-attach [COREID [COREID ...]]] [--profile]
                        [--host PROGRAM [ARG ...]] [--extra-args ARGS]
                        [PROGRAM [PROGRAM ...]]
```

Epiphany simulator frontend.
Helps spawning simulator processes.

Default configuration is Parallella-16 Epiphany-III.
Rows: 4 Columns: 4
External RAM size: 32 MB External RAM base: 0x8e000000
First core: 0x808 (32, 8)

Press Ctrl-C at any time to abort the simulation.

positional arguments:

PROGRAM	Executable program(s). Not required when environment is set to operational. Program(s) will be distributed to cores from left (west) to right (east) and then wrap to next row. If the number of programs is less than the number of cores, the last program will be used for the remaining cores
---------	---

optional arguments:

-h, --help	show this help message and exit
--verbose	Verbose output
-r ROWS, --rows ROWS	Number of rows
-c COLS, --cols COLS	Number of columns
-f FIRST_CORE, --first-core FIRST_CORE	Coreid of upper leftmost (northwest) core
-i FIRST_ROW, --first-row FIRST_ROW	Row of upper leftmost (northwest) core
-j FIRST_COL, --first-col FIRST_COL	Column of upper leftmost (northwest) core
--environment ENVIRONMENT	Environment. Must be one of 'user', 'virtual', or 'operating'. NOTE: Default is 'virtual'. This is different from epiphany-elf-run, where the default environment setting is 'user'
--ext-ram-size EXT_RAM_SIZE	External RAM size in MB
--ext-ram-base EXT_RAM_BASE	External RAM base address
--redirect-dir REDIRECT_DIR	Redirect stdin, stdout, and stderr to per process files in REDIRECT_DIR. REDIRECT_DIR will be created if it does not exist. Files in the directory will be overwritten. stdin will be redirected to REDIRECT_DIR/core-0xXYZ.stdin if that file exists.
--wait-attach [COREID [COREID ...]]	

	Will not spawn simulator processes for given COREID(s). This is useful when you want to attach cores through gdb
<code>--profile</code>	Enable profiling
<code>--host PROGRAM [ARG ...]</code>	Start simulation with host program. First argument is the native program to spawn. The following arguments are arguments to the host program.
<code>--extra-args ARGS</code>	Pass ARGS to <code>epiphany-elf-run</code> . Enclose in quotes, e.g <code>--extra-args="--foo --bar"</code>