# Bliss Hacking Guide

Bliss is a Modular Simulator

Steven Swanson and Mark Oskin
*swanson@cs.washington.edu, oskin@cs.washington.edu*

The Bliss architecture simulator toolkit is comprised of over 13,000 lines of C++ code spread across over 100 source files. This guide is an overview of how to hack the code to implement your particular hardware model. We begin this guide with a high level overview of the simulator.

## 1 Simulation overview

### 1.1 Core Abstractions

Bliss is comprised of three main abstractions: Engines, Components, and Channels.

#### 1.1.1 Engines

An *Engine* is an object that drives simulation models. In Bliss, engines provide instances of classes called *Instruction*. The Instruction class is an abstraction of a generic RISC like instruction. Engines provide instances of this class on demand. In addition, engines support the redirection of an application (i.e. mis-speculation).

Currently, Bliss supports a direct-execution Engine of the Alpha AXP architecture. Binaries (Alpha/OSF/COFF or Alpha/Linux/ELF) can be loaded by this engine and interpreted. A limited (basic) set of system calls are implemented that allow for common user-mode applications (without networking) to run. The engine supports speculative execution with full mis-speculation effects.

#### 1.1.2 Components

A component is the core object of a simulation model in Bliss. Every simulation model is derived from a Component. Components provide a host of standard and custom interfaces, depending upon their derivation in the C++ class hierarchy. Specific component models will be discussed in later sections.

#### 1.1.3 Channels

A channel, which is also a type of component, is the means by which components communicate to each other when event-driven simulation is used. As will be discussed in the next section, Bliss supports a standardized event-driven communication mechanism between components over a Channel, and custom inter-component functional interfaces through the C++ class interface.

## 2 Simulation itself

Actual simulation in Bliss is a mixture of event driven and function based simulations. The key difference between event driven and functional simulation is that event driven simulators focus simulation around the

dispatching and processing of messages (events). These messages arrive into modules through callback functions, which in turn generate new messages that are subsequently delivered. In contrast, functional simulation is structured around a sequence of function calls. When a module needs information from another module a function is invoked in that module that processes the request and returns that information back to the calling module. This calling process is usually done in a straightforward manner using the built in function and calling semantics of the language.

The two types of simulators tend to have their respective pros and cons. Event driven simulators tend to be slower than functional ones, and produce more obtuse coding styles. Once finished and working properly, however, they tend to be more flexible. It is also easier to model more complex interactions with event driven simulators instead of functional ones. The reason is that with functional simulators, modules often need to perform calculations about the future behavior of a model and return a result based upon that future behavior (for example, it is typical in a function based cache memory simulator to process a load or store request completely through all levels of the cache hierarchy and then to return the result to the processor model, all from a single function call made from the processor). This need in functional based simulators to calculate the future makes the modeling of complex behaviors, such as bus contention, difficult. Conversely, with event driven simulators bus contention modeling often happens for free as a byproduct of the message dispatching logic.

Event driven simulators are plagued with their own problems. The interaction between models and message traffic is often extremely difficult to predict. When bounded resource models are used, which is the ideal case since they more closely match reality (i.e. not many real chips have infinite size buffers), poorly written or configured models can live-lock. It is often difficult to predict the behavior of a simulator model since the message traffic it processes can be in an order the designer did not originally think about. This ordering is controlled explicitly in functional simulators by the order in which the simulator calls the modules (for example, one often sees in functional processor simulators that the processor pipeline stages modules are invoked back to front, completion stage to fetch stage). Event driven simulators dispatch messages based upon a centralized queue, which often makes the ordering of message delivery unpredictable. Consequently, poorly written or configured simulator models can receive messages in orders they where not written to expect.

Bliss attempts to combine functional and event driven simulation. With certain components, such as a superscalar processor, functional simulation works well. Instructions proceed through the processor pipeline in such a regular fashion that there isn't the need for the dynamism of event driven simulation. Functional simulation is sufficient because the need to predict the future affects of a simulator module is minimal (if almost non-existent). In contrast, the memory system is rife with dynamic behavior. There is really no telling how long a memory request will take until it is processed and its interaction is modeled with all other memory requests. These interactions cause contention for resources at times that are extremely difficult to predict far into the future (on the time scale of a cache miss). For this reason BLISS uses event driven simulation for the memory system.

The two styles of simulation, event driven and functional are glued together in what Bliss calls a Domain. A domain roughly corresponds to a group of components that interact synchronously (functional simulation) and asynchronously (event driven simulation). Components can still interact if they are in different domains, but only asynchronously. The key to this gluing of functional and event driven simulation styles is the use of controlling components. These controlling components (which are invoked in an asynchronous like fashion) then invoke function based simulation procedures in sub-components. As a specific example, the SuperScalarProcessor class manages the functional simulation of the various pipeline stages.

Within a domain a component can be invoked for simulation in up to three different ways. First, it can be simulated in a fully event driven style through a function AsyncSimulateOneCycle. This function is invoked one (to many times) each clock cycle. The order in which components are simulated through this function is not determinable and should not affect the simulation results. A component can also be

simulated through a transition function SyncSimulateOneCycle. This function is called only once per cycle (at the start of a cycle) and the order in which component's are invoked is not determinable. Finally, a function SimulateOneCycle is used by functional simulation management components. The order in which this function is invoked is "well known", and controlled by a management class.

Components can support any or all of these simulation functions, although usually only one is implemented. The exception is those components at the boundary between event driven and functional simulation. For example the FetchStage module supports both the SimulateOneCycle (well-known ordering, functional simulation) and the AsyncSimulateOneCycle (no-known ordering, event-driven simulation) interfaces.

A fourth way for a component to be simulated also exists. A special type of component, called a Channel is used to deliver events between modules. When a message is delivered into a component, three things occur. First, that components WhenCanYouTakeThisMessage function is invoked. If all receiving components can accept the message then all component's ReceiveMessage handler will be invoked. Finally, depending upon a configuration value, each component will be added to a "live list" for that cycle for re-invocation of the AsyncSimulateOneCycle function. This re-invocation of the AsyncSimulateOneCycle function is controlled by a configuration parameter, discussed later (to improve simulation performance with only a marginal decrease in accuracy this re-invocation is typically disabled).

# A Files and directories

Bliss is broken up into a number of files and directories. We will briefly describe these here.

## A.1 Directories

- *Foundation*: Core C++ classes (arrays, stacks, fifos, hash tables, priority queues), command line parsing, tuple spaces, memory leak detection, fast-allocation mechanisms.
- *Engines*: Contains the abstract C++ interface for an engine. Particular engine types should be derived from it.
- *Engines/SimulatedAlpha*: Contains the Alpha/AXP OSF or Linux binary interpreter.
- *Engines/SimulatedAlpha/Tables*: This directory contains definitions for the Alpha/AXP instructions. Instructions, their opcodes and register usages are stored here.
- *Engines/SimulatedAlpha/Implementors*: This directory contains the information on how to execute the alpha opcodes.
- *Components*: All simulation models should be placed off of this directory.
- *Components/Core*: Core component models (Component, Channel, Port, Domain)
- *Components/Channels*: Channel implementations (bounded and unbounded).
- *Components/Processor*: Processor cores should be placed off of this directory.
- *Components/Processor/SuperScalarProcessor*: This models a modern 7 stage pipelined superscalar processor. The processor model is loosely based around the modern Alpha architectures (i.e. physical register file instead of re-order buffer based).
- *Components/Memory*: This contains simulation models of clocked memory busses, caches, and DRAM components, as well as an overall memory system component.

- *Components/Systems*: In this directory are simulation models of system boards. A system board is really more of a container to bring together an engine, processor model, and memory system simulator.
- *Simulators*: This directory contains the front-end to the simulators (i.e. "main").
- *Configurations*: This directory contains configuration files.
- *bin*: Initially this directory is empty, but once you build the simulator the program ends up here.

## A.2 Files

- *Configurations/StandardConfig.cxx*:
- *Standard.h*: The include of all includes. All source files should start by including this file. This header file in turn, includes all other header files in the proper order.
- *Foundation/Global.h*: This file defines basic types such as uint64 in a cross-platform way.
- *Foundation/Compatibility.h*: This file contains any porting hacks. Namely, some hacks to make Bliss compile and work on Alpha/OSF boxes.
- *Foundation/Allocator.h*: The file contains a template class for the create of "free lists". Free lists help to speed up the allocation, deletion, and re-allocation of frequently used objects. Objects that are created often such as Message's and Instruction's should use a free list.
- *Foundation/Array.h*: This file contains a basic (growable) array template. It has all the usual features you would expect of such a class.
- *Foundation/Fifo.h*: This template implements a fixed size first-in first-out data structure.
- *Foundation/PQueue.h*: This template provides PQueue and PQueueOfPtr templates. These are auto-sorted data structures with Push insertion and Pop (lowest) element functionality.
- *Foundation/Stack.h*: This template provides LIFO (any size) capability.
- *Foundation/Hash.h/.cxx*: These files implement a hash table. Objects that go into a hash table should be derived from the HashableObject class.
- *Foundation/Debug.h*: This file just provides some macros that are used throughout the simulator for debugging purposes.
- *Foundation/TupleSpace.h/.cxx*: A tuple is a named (typed) object. Tuples of integers, booleans, and strings are supported. A TupleSpace is also implemented.
- *Foundation/CommandLineParser..h/cxx*: This implements the functionality required to parse command lines. It is not limited to such and can parse configuration files, and strings that contain one or more configuration options. These options are used to configure a TupleSpace.
- *Foundation/safemem.h/.cxx*: These are compiled in if DEBUG_MEMORY is defined. Note that when you compile Bliss in this way a lot of stuff will be spewed to the screen about memory, most of it being ok. The DEBUG_MEMORY flag is really only useful for catching memory leaks (of Allocator objects), and double free's.
- *Engines/Engine.h/.cxx*: This file contains the abstract definition of an engine. Not much code is here.

- *Engines/SimulatedAlpha/AlphaProcessor.h/.cxx*: This file is the main core of the Alpha AXP binary interpreter. The file is where all of the decoding tables are stored and the instruction implementations. Note, however, that the code for these is produced through a series of macros and inclusion of form/use/table files (see below).
- *Engines/SimulatedAlpha/Loader.h/.cxx*: These files contain the logic to decode COFF and ELF Alpha binaries.
- *Engines/SimulatedAlpha/Sparse64BitMemorySpace.h*: This file implements a sparsely populated 64 bit array. The array will be demand-allocated depending upon what you access. Clearly, on 32 bit systems you don't want to riffle through every possible address.
- *Engines/SimulatedAlpha/OSFSysCallDefinations.h*: This file defines the various OSF/Linux-Alpha system call numbers.
- *Engines/SimulatedAlpha/SystemTrapHandler.h/.cxx*: These files implement a basic subset of the OSF/Linux-Alpha system calls.
- *Engines/SimulatedAlpha/AllTables.h*: This file simply includes all known op-code tables.
- *Engines/SimulatedAlpha/AllForms.h*: This file includes all known forms. A form is a micro-environment in which an instruction implementation operates. The reason we use this is consider writing a simulator for the alpha add and subtract operations. The code would appear almost the same except for a $+$ instead of a $-$ in one spot. Thus, to short hand the writing of the AXP execution core the bulk of this code is encapsulated into a form. These forms are then instantiated by uses.
- *Engines/SimulatedAlpha/AllUses.h*: This file includes all known uses.
- *Engines/SimulatedAlpha/Tables/TopLevel.table*: See the Alpha AXP architecture handbook.
- *Engines/SimulatedAlpha/Tables/intm.table*: See the Alpha AXP architecture handbook.
- *Engines/SimulatedAlpha/Tables/fltv.table*: See the Alpha AXP architecture handbook.
- *Engines/SimulatedAlpha/Tables/fltl.table*: See the Alpha AXP architecture handbook.
- *Engines/SimulatedAlpha/Tables/misc.table*: See the Alpha AXP architecture handbook.
- *Engines/SimulatedAlpha/Tables/jmplink.table*: See the Alpha AXP architecture handbook.
- *Engines/SimulatedAlpha/Tables/flti.table*: See the Alpha AXP architecture handbook.
- *Engines/SimulatedAlpha/Tables/itfp.table*: See the Alpha AXP architecture handbook.
- *Engines/SimulatedAlpha/Tables/inta.table*: See the Alpha AXP architecture handbook.
- *Engines/SimulatedAlpha/Tables/intl.table*: See the Alpha AXP architecture handbook.
- *Engines/SimulatedAlpha/Tables/ints.table*: See the Alpha AXP architecture handbook.
- *Engines/SimulatedAlpha/Tables/fpti.table*: See the Alpha AXP architecture handbook.
- *Engines/SimulatedAlpha/Implementors/iOp3.form/.use*: These files contain the implementations for most 3 operand integer instructions. Miscellaneous instructions are also here.

- *Engines/SimulatedAlpha/Implementors/fOp3.form/.use*: These files contain the implementations for most 3 operand floating point instructions.
- *Engines/SimulatedAlpha/Implementors/iBr1.form/.use*: integer branch instructions.
- *Engines/SimulatedAlpha/Implementors/fBr1.form/.use*: floating-point branch instructions.
- *Engines/SimulatedAlpha/Implementors/iMem2.form/.use*: Integer memory I/O operations.
- *Engines/SimulatedAlpha/Implementors/fMem2.form/.use*: Floating point memory I/O operations.
- *Components/Core/Instruction.h/.cxx*: This is one of the central classes of Bliss. This class is used to store an instruction's information. An abstraction of a typical RISC instruction is stored, as well as a copy of the original Alpha AXP instruction. If you want to tag a piece of information to an instruction and have that be available to all the models of the processor, you stick it here in the Instruction class.
- *Components/Core/Component.h/.cxx*: This is a core foundation class of Bliss. All simulation models are derived from Component. Component provides the abstract interface to the simulation mechanisms, support functions for statistics and configuration output, and support for the hierarchy of components model used by Bliss.
- *Components/Core/ResourceLock.h*: This class encapsulates the basic working mechanisms of resource lock. Note, this is not a resource lock for writing mutlithreaded simulators. This is an abstraction of a resource lock used by simulation model.
- *Components/Core/ScheduledQueue.h*: This is a class, much like PQueueOfPtr, except highly optimized for near-term scheduling problems. It is used for scheduled access to resources (such as busses).
- *Components/Core/Domain.h/.cxx*: A Domain is a fundamental construct in Bliss. Components are members of a Domain. A Domain is a component (hence there is a hierarchy of domains). Domains are used to simulate multiple components.
- *Components/Core/Message.h/.cxx*: This is the base class for all messages. A Message is a special type of C++ object that is passed across busses. You must derive model-specific message types from this class.
- *Components/Core/Port.h*: A Port, separated into an InputPort and an OutputPort is used to manage the delivery of messages to components. Busses have one logical InputPort and one logical OutputPort. These logical ports are connected to 1 or more Component models, thus representing multiple physical ports.
- *Components/Core/MessageChannel.h/.cxx*: This is the base class for all message channels (busses).
- *Components/Channels/SimpleChannel.h/.cxx*: This is a simple channel that has no bounds on its communication amount.
- *Components/Channels/BoundedChannel.h/.cxx*: This is a bounded resource channel. It is used to model more realistic busses.
- *Components/Processor/SuperScalarProcessor/SuperScalarProcessor.h/.cxx*: The SuperScalarProcessor class is the C++ object that puts all of the processor stage modules together. If you want to insert pipeline stages, use your own stages, etc, then you will want to change the classes SuperScalarProcessor uses.

- *Components/Processor/SuperScalarProcessor/ProcessorStage.h/.cxx*: All processor stages are derived from the ProcessorStage class. This class provides the abstract mechanisms used to pass instructions between stages. The reason that an abstract base class is used here is so that you can insert additional pipeline stages, largely at will, without having to change the underlying models.

- *Components/Processor/SuperScalarProcessor/FetchStage.h/.cxx*: The FetchStage class is a complex block-orientated fetch stage model that interacts with a branch predictor, branch target buffer, and memory system. The FetchStage class will fetch whole block lines at a time, and has a configurable look-ahead mechanism.

- *Components/Processor/SuperScalarProcessor/DelayStage.h/.cxx*: This processor stage does nothing. It is meant to act as those stages of a processor that require only very simplistic software models (such as decode), or those stages of modern processors that are there simply for wire-delay. The DelayStage class is a configurable holder of instructions that is sufficient for these purposes.

- *Components/Processor/SuperScalarProcessor/RenameStage.h/.cxx*: After instructions are fetched they are sent to a decode stage (which is a DelayStage). From there they are sent to a rename stage. This stage renames the register inputs and outputs into a physical register pool.

- *Components/Processor/SuperScalarProcessor/IssueWindowStage.h/.cxx*: This stage is a holding bin for renamed instructions that are not yet ready to execute. They remain here until their dependencies are speculatively resolved. There is additional discussion later on this subject.

- *Components/Processor/SuperScalarProcessor/RegisterAccessStage.h/.cxx*: This stage modules the register access to a physical register pool. It sits between the IssueWindowStage and the ExecutionStage.

- *Components/Processor/SuperScalarProcessor/ExecutionStage.h/.cxx*: This stage modals the actual execution of instructions (or the address calculation of memory operations).

- *Components/Processor/SuperScalarProcessor/StoreBuffer.h/.cxx*: This stage modals a store buffer. Currently, load speculation is not performed. Rather memory operations are queued and then dispatched into the memory system such that total-load-store (TSL) ordering is maintained.

- *Components/Processor/SuperScalarProcessor/CompletionStage.h/cxx*: This is the final stage of the processor. This stage queues instructions and reconstructs program order. Possibly mis-speculated instructions are detected here and the FetchStage notified to reset.

- *Components/Processor/SuperScalarProcessor/Debugger.h/.cxx*: This is a debugger that is specific to the SuperScalarProcessor class. This debugger prints out instructions in a human-readable format and uses the data structures of the various processor stages to provide additional information, such as where the instruction is, what the state of its dependencies are, and how the processor stages are doing on executing it.

- *Components/Processor/SuperScalarProcessor/BranchTargetBuffer.h/.cxx*: This file provides an abstract interface to a branch target buffer (BTB) and two instantiations of it (a simple direct mapped, and a tagged BTB).

- *Components/Processor/SuperScalarProcessor/BranchPredictor.h/.cxx*: This file provides the BranchPredictor class, an abstract C++ interface class to the workings of a branch predictor.

- *Components/Processor/SuperScalarProcessor/GShareBranchPredictor.h/.cxx*: This is an implementation of the gshare branch predictor.

- *Components/Memory/MemoryTransactionMessage.h/.cxx*: The MemoryTransactionMessage class is a type of Message used for the communication of memory transactions (read, write, readresponse, shared memory, etc).
- *Components/Memory/SimpleClockedMemoryBus.h/.cxx*: This bus, derived from BoundedChannel, models a clocked memory bus that communicates MemoryTransactionMessage types.
- *Components/Memory/SetAssociativeMemory.h/.cxx*: This class encapsulates the logic behind a set associative memory. Used in Caches, types of branch target buffers, etc.
- *Components/Memory/SimpleCache.h/.cxx*: This models a lockup free cache with bounded resource constraints.
- *Components/Memory/SimpleSDRAMModel.h/.cxx*: This is a simple DRAM model with resource constraints.
- *Components/Memory/SimpleCacheMemorySystem.h/.cxx*: The SimpleCacheMemorySystem class puts together an SDRAM model, a unified L2 cache and split L1 instruction and data caches.
- *Components/Systems/SingleProcessorSystemBoard.h/.cxx*: This class puts together a processor model (SuperScalarProcessor) and a memory system model (SimpleCacheMemorySystem). It binds these together and provides support for simulating them.
- *Simulators/sim4.cxx*: This is the front-end (main) to the Bliss simulator. Here is where it all begins.
- *Simulators/run-alpha.cxx*: This is a fast, execution-only simulator. It is used just to quickly execute alpha binaries.

# B License