Jari-Matti Mäkelä | Jani Paakkulainen | Ville Leppänen

SMASim manual, version 1.0

Turku Centre *for* Computer Science

TUCS Technical Report
No 972, April 2010

SMASim manual, version 1.0

Jari-Matti Mäkelä
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, 20520 Turku, Finland
jmjmak@utu.fi
Jani Paakkulainen
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, 20520 Turku, Finland
janpaakk@utu.fi
Ville Leppänen
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, 20520 Turku, Finland
ville.leppanen@it.utu.fi

TUCS Technical Report
No 972, April 2010

Abstract

The computer industry has faced a constant need for more efficient hardware to perform computational tasks. Previous generations of microchips have tried to mitigate the problem on three fronts: increasing the execution speed by increasing the operating frequency, decreasing the amount of required time to issue a single instruction by enhancing *instruction level parallelism*, and increasing the "computational volume" by adding more computational units. Developing new architectures has turned out to be expensive and requires relatively great amount of resources even when building experimental FPGA prototypes.

SMASim is a software based simulator, motivated by an experimental moving threads architecture, that attempts to lower the costs of rapidly designing new architectures. It is based on a general purpose, precise latency centric message passing framework between the described hardware architecture elements. Its relatively simple cost model captures the essential properties of many hardware designs. The simulator's design allows easy monitoring of the system and provides execution performance comparable to other cycle-accurate hardware simulators. The implementation is cycle-accurate, modular, and supports simulation on various granularity levels.

The simulator is implemented in hybrid object-functional language Scala. The flexibility of Scala allows using declarative domain specific notation when specifying parts of the architecture, yet it provides a static verification of the model via a strong type system. A graphical user interface is provided to simplify the task of modifying parameters of a simulated system and to provide interactive feedback.

Keywords: processor simulator, cycle accurate, multi-core, moving threads, domain specific language

TUCS Laboratory
TUCS Algorithmics Laboratory

Contents

1 Introduction

As the performance requirements of modern software have steadily increased, the computer industry has faced a constant need for more efficient hardware to perform these tasks. Previous generations of microchips have tried to mitigate the problem on three fronts: (i) increasing the amount of executed instructions per time unit by increasing the operating frequency, (ii) decreasing the amount of required time to issue a single instruction (CPI) by enhancing *instruction level parallelism* (ILP), and finally (iii) increasing the "computational volume" by adding more computational units.

During the last few years the first two approaches have reached the point of diminishing returns. The operating frequencies have stabilized on the 2...3 Ghz range. Complex ILP mechanisms begin to occupy on-chip space in ways that make multi-core solutions appear more feasible. The amount of parallelism on instruction level is also limited by data dependencies. Another physical limitation worth mentioning is the power dissipation of the processor, which has increased over the years to the point that hardware reliability is endangered unless powerful cooling systems are utilized.

The third approach has started a new era of multi-core and multi-processor parallel programming. Several vendors have provided their first revisions of multi-core architectures. Efforts that stand out from the traditional *von Neumann model* seem to yield most promising performance improvements. Developing new hardware architectures has turned out to be expensive and requires relatively great amount of resources even when building experimental FPGA prototypes.

SMASim is a pure software based simulator, which attempts to lower the costs of rapidly designing new architectures by supporting new multi-core architectures at various abstraction levels. Even though SMASim was built for simulating special kinds of multi-core systems, its core is a general purpose framework, although the standard set of components has been tuned for a specific multi-core architecture.

SMASim uses a precise latency centric message passing system between stateful simulated components. Its relatively simple cost model captures the essential properties of most hardware designs, allows easy monitoring of the system, and provides adequate asymptotic execution performance whilst not resorting to dynamic translation of the architecture model. The implementation is cycle-accurate, modular, and supports simulation on various granularity levels.

The simulator is implemented in hybrid object-functional language Scala [8]. The flexibility of Scala often allows using declarative domain specific notation when specifying parts of the architecture, yet provides static verification of the model via a strong type system. The name SMASim has two meanings — the simulation is built on Scala and also adapts the Scala's idea of a scalable language. Systems built on SMASim have the flexibility of a modern general purpose language, but also allow defining parts of the system in a declarative manner with a

custom DSL (domain specific language).

The core framework and building blocks of the simulator are tuned for high performance and to minimize memory allocations. However due to limitations of Java virtual machine and strict cycle-accurate execution, some performance is sacrificed for other high level features.

Although having a generic CPU simulation framework is a worthy goal per se, the main motivation behind SMASim was to provide an exact software implementation of the moving threads architecture ([10]) that is based on a virtual *parallel random access machine* (PRAM) [3, 5] model and differs from contemporary designs regarding the thread abstraction and handling. The architecture is a new kind of multi-core on chip processor which differs from contemporary designs regarding the thread abstraction and handling.

However, the currently available version of SMASim has been extended and generalized to support other kinds of systems as well. All of the framework's modules now have clear roles and support interoperation on all abstraction levels. A graphical user interface is also provided to simplify the task of modifying parameters of a simulated system and to provide interactive feedback.

The current set of features in SMASim support in determining efficient parameters for various memory subsystems such as registers and caches, execution pipeline, and inter-processor network. Some properties, such as the supported instruction set, are more laborious to alter due to the inherent design complexity and are only briefly considered in this manual.

2 Related work

Architecture simulators are a widely studied subject. The implementations can be roughly split into two categories, based on their focus on functionality or performance. Most of the existing work encompasses a simulator core built on some high-performance, low-level systems programming language and a target architecture description either written in some declarative markup language or using the same core language. Typically, the simulators have mainly focused on existing commercially available single-core architectures.

2.1 Cycle-accurate simulators

The list of traditional cycle-accurate simulators is long. Probably the best known simulator is the SimpleScalar [1] project, a basic simulator with a straightforward implementation, which has already reached its fourth version. SimpleScalar allows defining single-core system with a definition language. Another widely used simulator is SESC [9], which uses C++ to model the architecture. PTLSim [11] and Bochs [7] simulate the x86 architecture.

2.2 New generations

More recently, the focus has been on more efficient simulations that either sacrifice the cycle-accuracy by introducing some statistical measurement based on random samples or implement advanced techniques for speeding up the event based system.

For instance, FaCSim [6] uses three methods to increase its speed: chunked pipeline events, caching of decoded instructions, and parallelization of the simulator. A lesser known simulator, tsim, takes advantage of a mechanism called two-phase trace-driven simulation (TPTS) [2]. The TPTS technique improves both, the execution speed and the memory consumption.

3 Overview

The simulation framework consists of several independent modules. The overall structure of these modules is described in the following Sections 3.1 . . . 3.7.

3.1 Core module

The core module contains necessary components for modeling generic message passing systems. The major components of this module are described in more detail in Section 4. Various kinds of designs are possible and the module is not limited to physical electronic components such as logical gates.

The module also contains various helper classes, e.gḟor logging the state of the simulation and modeling machine addresses and data.

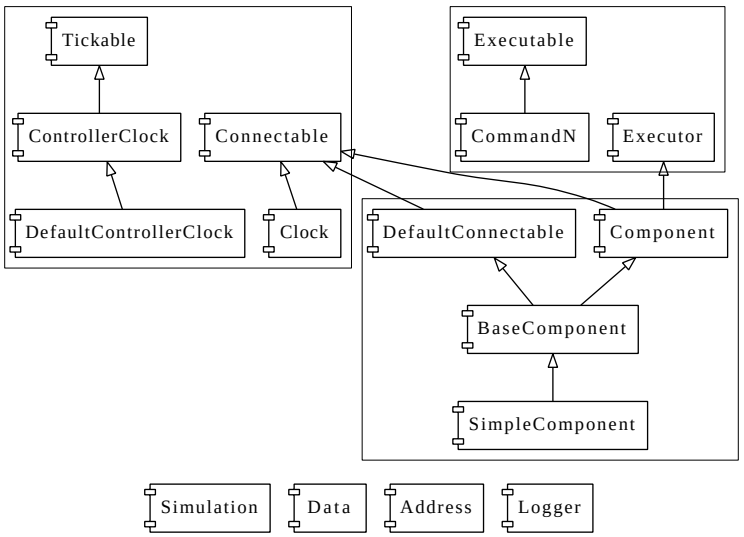

Figure 1: Class structure of the core module.

3.2 CPU module

The CPU module, discussed in more detail in Section 6.1, provides basic components for modeling central processing units and their internals, the basic building blocks of computer architectures. The basic structures include e.ginstructions, instruction sets, and means for importing program data from the filesystem.

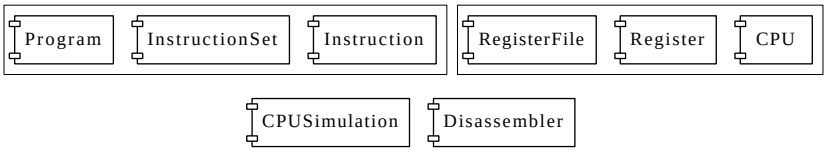

Figure 2: Class structure of the CPU module.

3.3 Memory module

The memory module contains memory and cache classes. The default memory component abstraction provides a flat address space and fixed (expectationbased) latency cost. Another basic memory component, an n-way associative cache abstraction, is suitable for simulating various kinds of practical memory hierarchies. Specialized memory abstractions can be further derived from the built-in components. The reader is adviced to read Section 6.2 for more information on the built-in memory components and their constructor interfaces.

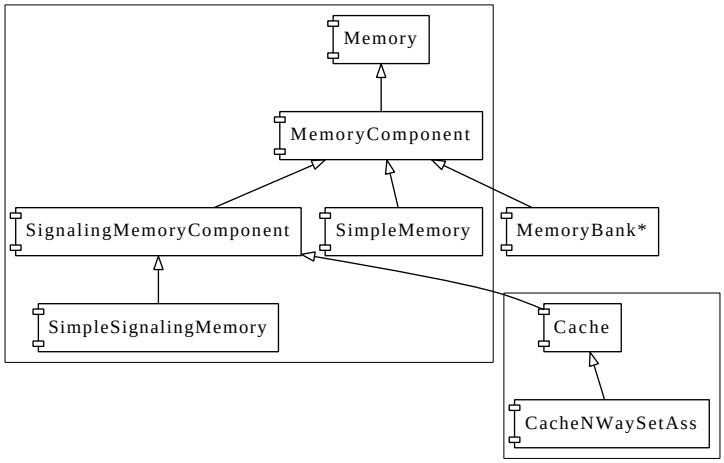

Figure 3: Class structure of the memory module.

3.4 Network module

The network layer has very simple generic interfaces for black box networks. The components attach to the message passing framework and provide a simple communication network for simulated systems. All network components are discussed in more detail in Section 6.3.

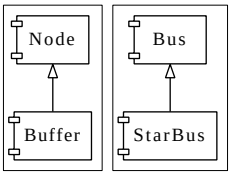

Figure 4: Class structure of the network module.

3.5 MIPS32 module

The MIPS32 module was the first instruction set implementation provided alongside the simulation framework. It can found useful both when implementing a MIPS32 style architecture or as an inspiration for implementing other instruction sets. Section 6.4 treats the MIPS32 layer along with examples that combine MIPS32 classes with functionality from the CPU module (Section 3.2).

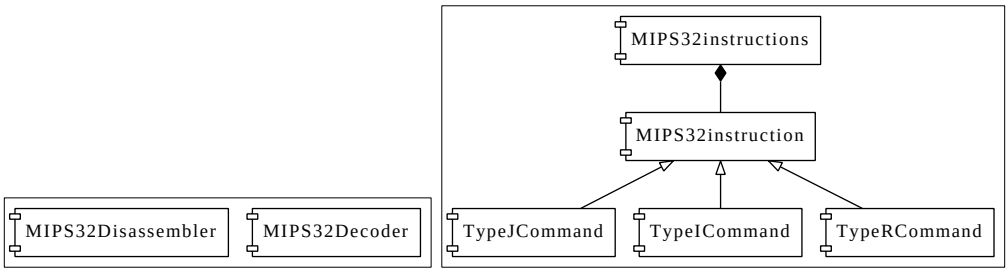

Figure 5: Class structure of the MIPS32 module.

3.6 Monitoring module

The monitoring module is a generic framework for monitoring simulated systems. It is an independent subsystem that connects to the simulation via the low-level message passing system. The monitor architecture and few examples of monitors are introduced in Section 5.1.

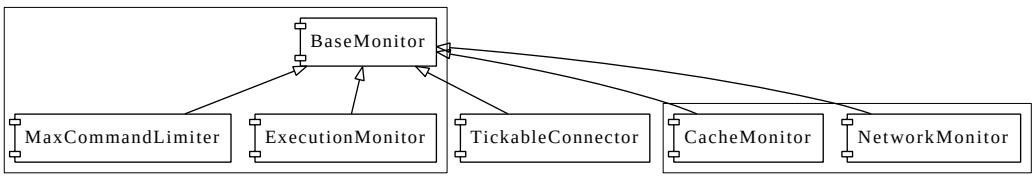

Figure 6: Class structure of the monitoring module.

3.7 GUI module

In addition to simulation related low level components, a system for visualizing the simulation is provided. The basic setup provided by the framework contains visual components for e.gcontrolling the simulation in a simple manner, displaying its textual state, and importing files from the filesystem. A more thorough treatment of the GUI module is given in Section 5.2.

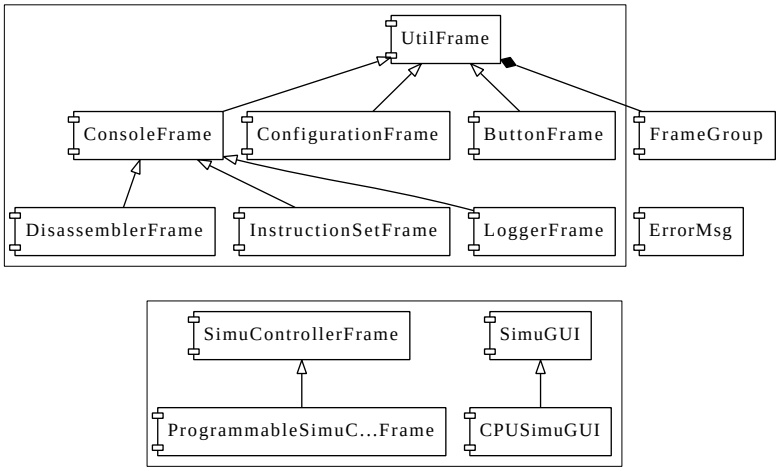

Figure 7: Class structure of the GUI module.

3.8 Module relations

To better illustrate the dependency relations between the modules, the modules adhere to the graph described in Figure 8. The graph can be used to determine which parts of the framework also may need to be built and imported when using a module.

The modules for GUI and monitors present cross-cutting concerns, and thus can be split further. In practice, the relevant part of the these modules is automatically imported when a set of built-in modules is being used in a project.

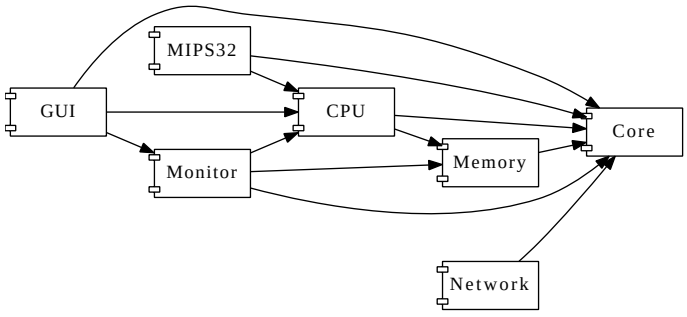

Figure 8: Module dependencies.

The Figure 9 shows a typical high level collaboration relation graph between various components in an imaginary simulator configuration. The graph has been split into three parts, the generic simulator logic provided by the simulation framework, the graphical user interface (GUI), and the system to be simulated.

All connections between modules in the system to be simulated depend on the actual hardware architecture, and thus are not discussed here in more detail.

The monitors attach to the system to be simulated via the controller clock mechanism. The mechanism provides a trace of all messages passed in the system.

The GUI components mainly connect to the simulation instance, the hardware architecture components, the monitors, and few internal framework components such as the logger. The purpose of the GUI is to provide a real-time view of the simulation and also to allow controlling it.

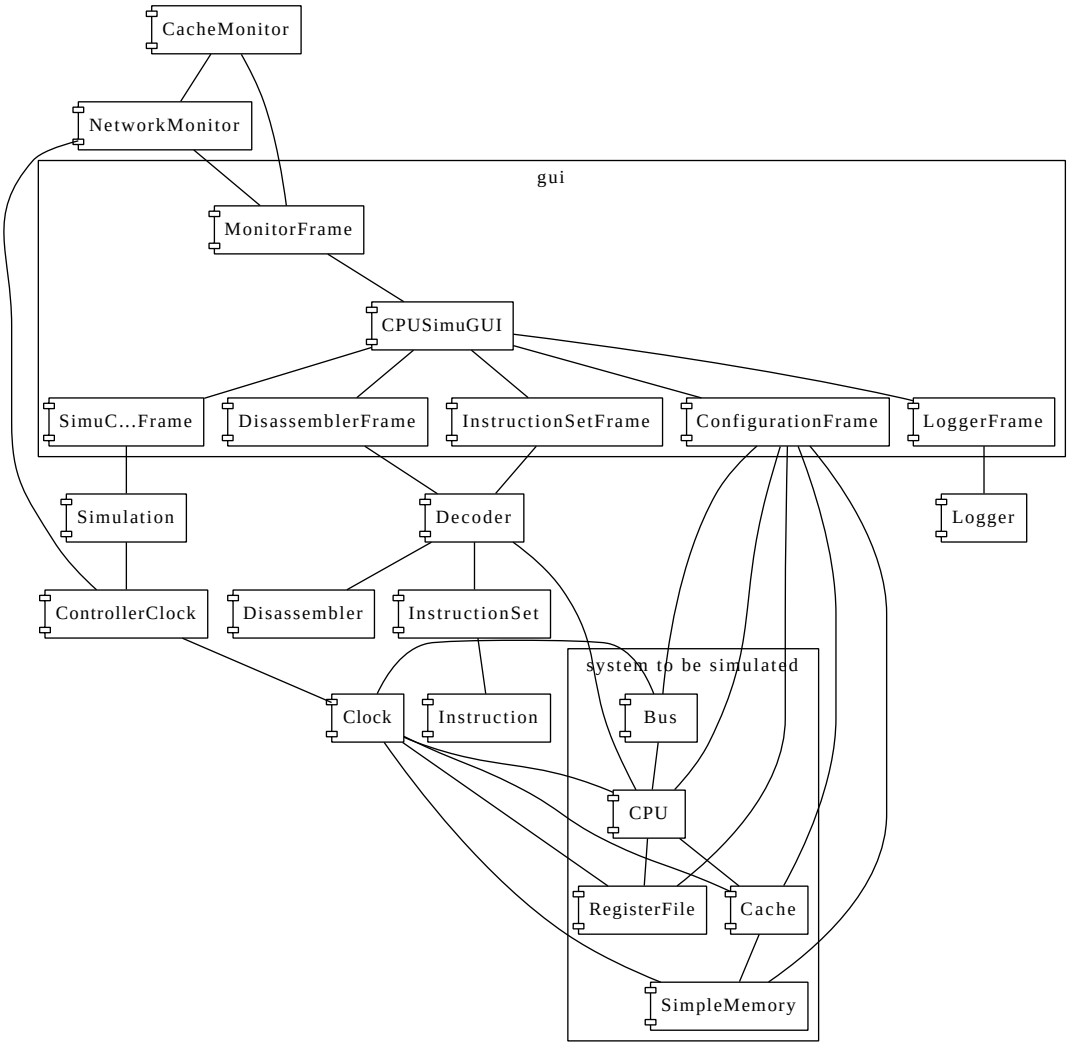

Figure 9: Module usage example.

4 Message passing architecture

4.1 Introduction

The core of the simulator is comprised of a strongly typed, latency-aware, event-driven message passing framework. All simulated computations are modeled as a static set of stateful processes (called commands), process executors, and stateless messages [1] with pre-defined latencies. In addition, all computations are triggered by logical clocks, which form the fourth category of core components.

Processes can be seen as a generalization of the GOF Command pattern [4] or as functions operating on multiple inputs with varying latencies and providing a single output. The event-driven execution strategy is beneficial when the number of simultaneous messages is much smaller than the number of components.

To better illustrate how the written specification maps to real world components, we continue with the implementation of a simple unit time logic gate AND presented in the Figure 10.

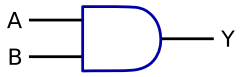

Figure 10: The AND gate.

The execution of the process is triggered once both inputs A and B have been received (possibly in arbitrary order) and the associated clock ticks. When compared to simulation designs which use a wire based abstraction, the control flow does not need to be explicitly defined with logical circuits, since the simulation is driven by the data flow. This somewhat changes the semantics compared to equivalent concrete implementation, but the solution simplifies the simulation a bit.

Algorithm 1 [2] describes the simulator code for the gate. The example begins by defining a clock with a fixed operating frequency (100 Hz). The clock connects to a global controller clock that operates as a master clock by running several clock sources with varying frequencies in synchrony. The `Clocks` merely act as a clock rate divisor, and the controller clock is used to trigger the execution. Several simultaneous controller clocks and clock types are supported, but the default global instance `ControllerClock` is used to avoid extra verbosity.

[1] Technically the model could be modified dynamically, but the corresponding actual hardware implementations rarely are capable of performing self-modification.

[2] The code examples in this manual use the literate programming style which replaces commonly used programming language tokens such as => and <- with ⇒ and ←. Whenever a non-ASCII character is encountered in the code, a C language style ASCII sequence should be used instead.

Algorithm 1 Implementation of the AND gate.

```
// initialization (ControllerClock & SimpleComponent are explained in more detail later)
implicit val clock = ControllerClock @> 100 Hz

class LogicalComponent extends SimpleComponent { type B = Boolean }

class AND extends LogicalComponent {
  def execute = value { (A: B, B: B) ⇒A & B }
}
```

Building a functional system from components is most often done via composition. Once the abstraction is defined, the usage turns out to be rather straightforward. For example, computing the logical AND of four inputs using dual input AND gates can be achieved by chaining the gates as shown in Figure 11.

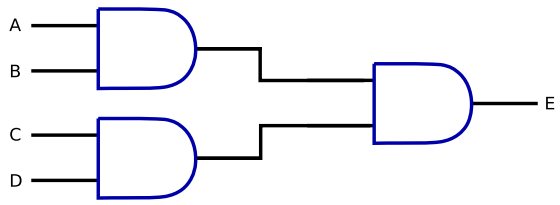

Figure 11: Composition of three AND gates.

The corresponding code is shown in Algorithm 2. A tuple of three AND components is first constructed. The connection between gates is added with the `=>:` notation. The problem with this approach is that all AND gates are still distinct objects and there exist no entity with four inputs, which makes it impossible to treat the circuit as a single object.

Algorithm 2 Implementation of the complete circuit.

```
def createAND = new AND execute // creates a new 2−port AND instance
val (and1, and2, and3) = (createAND, createAND, createAND)

(and1, and2) ⇒: and3
```

In modular systems a composition is accompanied by encapsulation. Conceptually we often want to hide the implementation details from the public interface of the component, as in object oriented or functional programming. To achieve this, the simulator supports encapsulating commands inside other commands.

The composition of three AND gates is visualized in Figure 12 and implemented in Algorithm 3. Hiding the complexity this way introduces a scalable way

of modeling larger systems with modular building blocks. This approach also allows mixing components modeled on different abstraction levels.

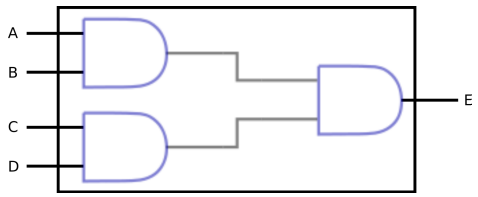

Figure 12: Encapsulated composition of three AND gates.

Algorithm 3 Implementation of the circuit with encapsulated three AND gates.

```
class Four_Port_AND extends LogicalComponent {
  def execute = fun { (a: B, b: B, c: B, d: B) ⇒
    ((a,b) ⇒: createAND,
     (c,d) ⇒: createAND) ⇒: createAND
  }
}
```

This cursory introduction gave a vague idea of the ways of using the simulation framework for building e.g̈logical gates and more complex logical circuits. The key ideas are modeling the components comprising the system, and mapping the internal logic of the components to available programming language's (i.e. *Scala's*) constructs. The same core features of the framework are discussed in more detail in latter parts of this section.

The rest of the section discusses the message passing system in more detail. A similar introduction to defining high level abstractions is presented in Section 7.

4.2 Executable

The `Executable` is a common interface for all messages passed in the system. As mentioned in Section 4.1, the simulation framework is an event-driven system with a message/process model. All simulation events are first initialized by enqueuing the descendants of the `Executable` interface (Algorithm 4) to an event queue. The queues are located in `ControllerClocks` and can be accessed via `Clocks` as described in Section 4.4. Later, the internal clock logic issues the commands according to their timestamps.

Thus, the low level interface provided by `Executable` is mainly relevant only when manipulating the existing message passing framework or when monitoring event execution. For practical applications it is recommended to build the component functionality on the deriving `Command` classes, which provide interfaces for the functional representation of the processes.

Algorithm 4 Executable interface.

```
trait Executable {
  def execute: Unit
  def executor: Executor
  def reset: Unit

  def description: String
  def name: String
  def latency: Int

  def @@(l: Int): this.type // set latency
  def @@(s: String): this.type // set name
}
```

4.2.1 Commands

`Commands` are convenience classes that implement the logic for initializing processes, defining their latencies, attaching them to some executor and logical clock, handling the arbitrary data dependencies between commands, and for implementing higher order commands (see Algorithm 3).

Algorithm 5 Simplified interface of Command, Value, and Fun.

```
abstract class CommandN[T1, ..., TN, TR] {
  ...
  def compute(v1: T1, ..., vn: TN): TR
}

object Value {
  def apply[T1, ..., Tn, R](e: Executor, v: (T1, ..., Tn) ⇒R): CommandN
}

object Fun {
  def apply{[T1, ..., Tn, R](e: Executor, v: (T1, ..., Tn) ⇒Returns[R]): CommandN
}
```

When building new components, the `Command` classes can be extended directly by providing a `compute`-method (Algorithm 5). An alternative method is to pass a lambda to one of the convience methods (`value`, `fun`, and `cmd`) in `Executor` objects (Algorithm 6) or to use the global singletons Value and Fun. These methods construct a new Command object with the provided `compute` method, and attach the command to the executor object. The method `value` must return ordinary values, `fun` other commands, and `cmd` either values or commands

wrapped in `Value` or `Fun` types, respectively. Algorithms 1 and 3 demonstrate the use of these methods.

In addition, commands can be chained according to their functional dependencies. Command classes provide :⇒ and ⇒: operators for expressing the dependency relations. The :⇒ form is used, when the left hand side is another command, and the latter form, when the left hand side is a hard coded value, a tuple of values, or a tuple of commands. The latter form was used in defining dependencies in Algorithms 2 and 3.

When a component does not determine a single common latency for all associated commands, a per-command latency can be defined with `the @@` method. This latency can be dynamically re-set at any time before the initialization of each execution. To conserve memory, the same commands can be reused by re-setting their state between executions of the event. In fact, this is done automatically by default.

4.3 Executor

`Executor` (Algorithm 6) represents the active components in the simulation. They define the means for enqueuing `Executables` and determining their latencies when executed by this particular executor.

Algorithm 6 Executor interface.

```
trait Executor {
  def latencyOf(executable: Executable): Int
  def enqueue(executable: Executable): Unit

  def name: String

  final def value[...](t: * ⇒*)
  final def fun[...](t: * ⇒*)
  final def cmd[...](t: * ⇒*)
}
```

Semantically `Executor` is a view of the logical entities in the simulation from the message passing system's point of view. Other aspects of the entities such as the physical connections are split into `Connectable` and other interfaces. As previously, the main functionality of most logical entities has been compiled into convenience interfaces, which are `Component` and its descendants. Executor also turns out to be most useful when monitoring events or rewriting the message passing logic.

Data can be passed between the executor entities with commands, which have similarities with first class functions in programming languages. The use of helper

functions `value`, `fun`, and `cmd` (described in Section 4.2.1) are recommended over direct accesses to the `Command` classes.

4.3.1 Component

`Component` (see Algorithm 7) ties together the executable process logic, its latency characteristics, information about the static component interconnection network, the associated clock that is used to trigger aforementioned events, and limits on the amount of simultaneous executions within the executor instance. Additionally, few other component interfaces are provided for stateful and storage components. It is recommended to derive all simulated entities from these interfaces or their descendants, utility classes such as `BaseComponent`.

Algorithm 7 Component interface.

```
trait Component extends Executor with Connectable {
  def maxCommandCount: Int
  def clock: Clock
}
```

Since much of the basic component functionality works in a similar fashion in all components, `BaseComponent` (Algorithm 8) alleviates the problem by implementing most of the low level book-keeping tasks. For even simpler components with fixed command latencies, there is a small wrapper, `SimpleComponent` (Algorithm 8), which is derived from `BaseComponent`.

A person well versed in Scala's syntax might notice that the clock instance can be implicitly provided on the call site. As one can see from Algorithm 1, the mere existence of a clock object in the scope suffices to construct a component object. This feature helps separating crosscutting concerns, since often components of the same module also share the clock instance. An explicit clock instance can always be provided.

Algorithm 8 Additional basic component interfaces.

```
abstract class BaseComponent(maxCommandCount: Int = 1)(implicit clock: Clock)
    extends StatefulComponent with DefaultConnectable

abstract class SimpleComponent(latency: Int = 1, maxCommandCount: Int = 1)
                              (implicit clock: Clock)
    extends BaseComponent(maxCommandCount)(clock)
```

With the helper classes, only the concurrent command execution upper limit and the clock entity need to be specified when constructing components. In prac-

tice, most of the higher level components (see Section 6) provided by the SMASim framework are built on top of these classes.

4.4 Clocks

The framework contains two kinds of clocks. The `ControllerClock` (see Algorithm 9) is a central entity for coordinating logical clocks with varying operating frequencies. For example memory modules and processing units in a computer often do not share the same operating frequency. A distinct clock entity can be used for each component type.

Usually a single controller clock is sufficient in a single simulation, but in special cases several instances can be used independently. Thus a global instance is provided for the common use case, as shown in Algorithm 2.

Algorithm 9 ControllerClock interface.

```
trait ControllerClock extends Connectable with Tickable {
  def add(clock: Clock): ControllerClock
  def enqueue(executable: Executable, latency: Int): Unit
  def executeCommand(executable: Executable): Unit
  def getNextCommand: Executable
  def updateCommands: Unit
  var canExecute: Boolean
  def @>(freq: Int) = new Clock(freq, this)
}
```

A fixed size two dimensional ring buffer is used as an executable buffer to minimize memory allocations during the simulation. The buffer has two important parameters: maximum executable count per time unit and maximum executable latency. Both need to be adjusted upfront to be large enough or the simulation will run out of resources.

The global `ControllerClock` instance supports enqueuing 100 concurrent executables on the same time unit, and a maximum latency of 512 time units. If different parameters are needed, a new `ControllerClock` instance has to be made as described in Algorithm 10. The maximum latency is required to be a power of 2 if the default implementation of `ControllerClock`, located in the class `DefaultControllerClock`, is used.

Algorithm 10 Constructing a new controller clock.

```
val cclock = new DefaultControllerClock(
  simultaneousMsgs = 1000,
  maxLatency = 512
)
```

`Clock` entities (see Algorithm 11) act as proxies to the associated, shared `ControllerClock` instance. The latency characteristics are modified automatically when enqueuing to compensate variations in execution frequency. Most of the clock logic is executed when an executable is enqueued; the actual executable latency is fixed at this point of time and the executable's position in the internal buffers is computed. In practice, the static dependencies defined with the arrow methods in `Commands` hide much of the bookkeeping required. The clock system also only needs a single link point to the components, which is usually done statically and implicitly when constructing new components.

Algorithm 11 Clock interface.

```
class Clock extends Connectable {
  def freq: Int
  def controller: ControllerClock
  def divideBy(div: Int): Clock
  def enqueue(e: Executable, latency: Int): Unit
  def add(c: Connectable): Unit
}
```

After enqueuing, the executable has been stored to the buffer and is waiting for execution triggered by the `tick` routine. The execution is triggered once the tick has been called as many times as is the command latency.

If the latency is set to 0, the executable will be executed some time after the current executable during the current call of the `tick` routine. In general, the system cannot guarantee any kind of ordering among messages to be executed during the same time step, if all data dependencies are met. For instance in the introductory example, with the four-port AND gate, the execution of the first two gates is undeterministic. However, the current implementation guarantees that two simulation runs always issue the commands in the same indeterministic order.

In a cycle accurate architecture, deterministic sequential order between events can be achieved by either adding enough delay to force event execution on different time units or by chaining executables via data dependencies.

In SMASim a flexible message passing architecture is used for modeling the target architecture. This decision allows composing the design with modular building blocks and simultaneously on variable abstraction levels. For instance, some components can be modeled on the level of logic gates while some other components are still huge black boxes with nontrivial functionality not directly mappable to physical components.

4.5 Connectable

Connectable (see Algorithm 12) is a utility interface that is implemented by all `Components`. It can be used for debugging and visualization purposes, since it maps a human readable name, a description, and a static list of connections (dependency or composition) to each component. By monitoring `Executables` and extracting static configuration from the `Connectables`, both dynamic actions and static relations and dependencies between components can be traced. Again, a higher level interface `DefaultConnectable`, which simplifies defining connections, is provided for convenience.

Algorithm 12 Connectable interface.

```
trait Connectable {
  def name: String
  def connections: Seq[Connection]
  def description: String
}
```

5 Utility layers

5.1 Monitoring framework

A CPU simulator is not only useful for revealing the outcome of an application written for the target architecture. For some applications a partial or full memory snapshot after the program execution suffices, but there also exist various emulators that provide a virtual terminal or graphical communication interface to the target architecture to e.g. provide an access to non-native operating systems.

The built-in components of the framework do not provide these communication layers yet, but instead a set of low-level monitors for monitoring the runtime behavior and collecting statistics are implemented. The main motivation behind the existing monitors was to ease the evaluation of the simulated architecture and to better tune the CPU parameters. The currently available monitors are tuned for command line output, but with small modifications they also can be used in GUI applications.

5.1.1 Execution monitor

A very basic monitor (`ExecutionMonitor`) for monitoring execution of various commands is provided in the monitoring framework. It is a very low level mechanism and is mostly useful for constructing other monitors or debugging the low level operation of the framework or the system used in the simulation.

The monitor works by connecting to a controller clock as a proxy. The simulation should then use the monitor in place of the original controller clock. Several monitors can be chained in this fashion as they are fully transparent. A logical setup with one execution monitor is described in Figure 13.

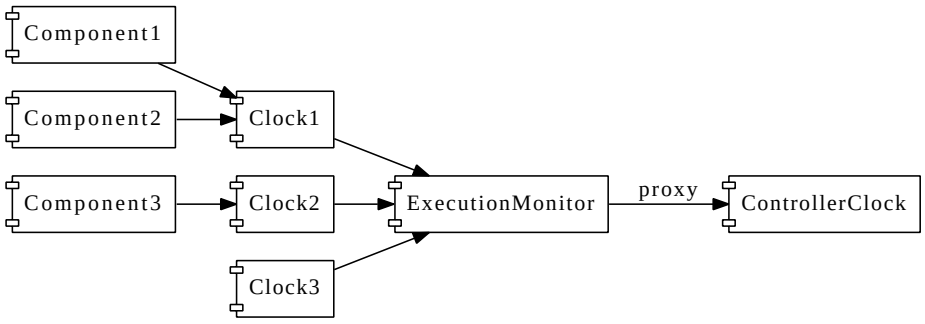

Figure 13: Simple execution monitor setup.

The default execution monitor accumulates a textual execution history based on the captured events. It extends the `GenericControllerClockMonitor`, an abstract class that fuses `ControllerClockProxy` and `Tracer` (see Algorithm 13). `ControllerClockProxy` implements simple functionality to

proxy a controller clock behavior transparently. `Tracer` provides a new interface for monitors that trace events such as enqueuing and execution of executables, and controller clock ticks.

Algorithm 13 Simplified tracer interface

```
trait Tracer extends ControllerClock {
  protected def traceExecution(e: Executable): Unit
  protected def traceTick: Unit
  protected def traceEnqueue(e: Executable, latency: Int): Unit
  ...
}
```

Usage of execution monitors is straightforward: The monitor object is constructed with the previous controller clock or monitor in the chain as parameter. A simple monitor configuration with a single controller clock and an execution monitor is described in Algorithm 14.

Algorithm 14 Usage of a simple execution monitor.

```
val monitor = new ExecutionMonitor(ControllerClock)

/* more initialization here */

while (monitor.isActive) { monitor.tick }
println(monitor.toString)
```

5.1.2 Derived monitors

The message passing system forms the main communication infrastructure inside the simulator. It separates functional units (components) from each other. One can always build custom mechanisms for monitoring programs, but the generic execution monitor (Section 5.1.1) interface often suffices. One advantage of the built-in monitoring system is that it is pluggable and thus also separated from other concerns in the simulator design.

As we can see from the Tracer interface in Algorithm 13, the execution monitor is triggered by three kinds of events: clock ticks, executable enqueuing and execution. In addition to passing messages, the components can have an internal state for e.g. monitoring purposes. If this interface is used for monitoring components, the component model should be fine-grained enough for the monitoring system to be able to capture relevant information during the simulation.

Usually the `tick` interface is used to determine, how many cycles have elapsed since the last timestamp or since the beginning of the simulation. On the other

hand, the pattern matching mechanism can be used to pick messages relevant to the monitor from the stream of new events.

As an example of using the monitor interface, two monitors (Sections 5.1.2 and 5.1.2) for the built-in components described in Section 6 are described next. Both monitors extend the generic `GenericControllerClockMonitor` and connect to a previous controller clock, either the actual controller clock or some monitor that works as a controller clock proxy. The output accumulates during execution and can be read using the `toString` method.

Cache monitor The cache monitor can be attached to a memory cache unit. During the simulation cache monitor collects information about cache accesses, hits, and misses. Cache hit ratio and latency characteristics are computed from the monitored data at any point of time. Cache statistics are also collected from the beginning of the simulation until the last status query. All caches (see Section 6.2) connected to the controller clock will be monitored and the output will group the cache events by a cache name.

Network monitor A network monitor traces the packet traffic on the inter-processor network or more generally any kind of data moving within a generic network abstraction. The latencies of packet transmissions and amount of packets and bytes transferred from node to node are being monitored. All networks implementing the network abstraction (see Section 6.3) connected to the controller clock will be monitored and the output will group the network traffic by a network name.

5.2 GUI framework

The GUI framework contains components for visualizing and controlling the simulation. The core GUI library consists of the following components:

1. A common control panel for all kinds of simulations is provided by classes `SimuGUI` and `SimuControllerFrame`.
2. An utility class for displaying error messages (`ErrorMsg`).
3. A class for showing menus comprised of buttons (`ButtonFrame`).
4. A generic configuration visualization and synchronization class (`ConfigurationFrame`).
5. Classes for displaying text, monitor, and log output. (`ConsoleFrame`, `MonitorFrame` and `LoggerFrame`).
6. Generic template class for other frames (`UtilFrame`).
7. Classes for grouping windows (`FrameGroup`).

5.2.1 Simulation controllers

SimuGUI The top level simulation controller frame (Figure 14) connects the simulation to the framework's GUI. By default controls are provided for:

- launching the simulation,
- shutting down the simulator,
- loading binaries from the file system,
- configuring the simulation parameters,
- displaying disassembler output,
- displaying system log,
- displaying supported instruction set, and
- displaying generic benchmark screen (less meaningful).

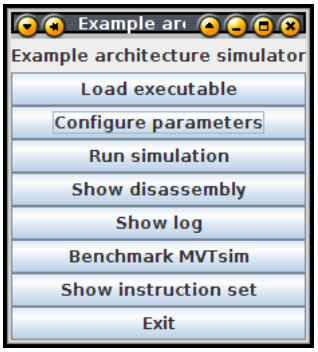

Figure 14: Main menu of the simulation GUI.

SimuControllerFrame The simulation controller frame (Figure 15) opens when the simulation has been started and closes when returning from the simulation to the top level menu. By default, the frame has support for:

- resetting the simulation,
- stepping forward 1, 10, . . . , 10000 time units, and
- grouping subwindows such as monitors.

Figure 15: Simulation controller frame.

5.2.2 Helper classes

ErrorMsg The error message dialog (Figure 16) can be used to display informational error dialogs in unexpected situations.

Figure 16: Error message dialog.

ButtonFrame A button frame presents a frame consisting purely of buttons. It is useful for creating e.g.simple controllers. Figures 14 and 15 show common instantiations of button frames.

ConfigurationFrame A configuration frame (Figure 17) presents a configuration screen for any aggregate type using Java's reflection system. Very useful when setting up simulation parameters.

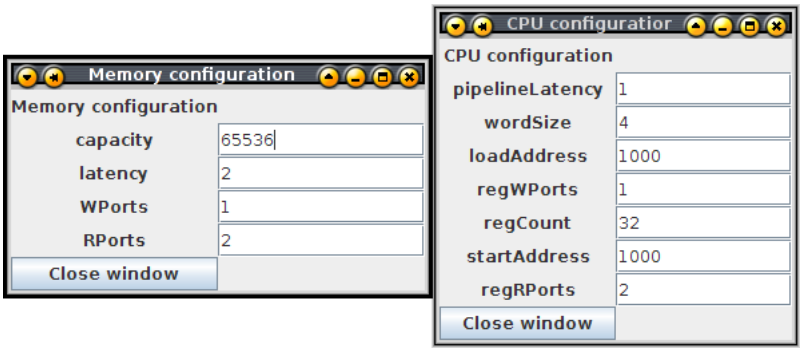

Figure 17: Configuration frame.

ConsoleFrame The console frame (Figure 18) shortens the time to create new frames consisting mostly of textual information. MonitorFrame and LogFrame both internally create a ConsoleFrame.

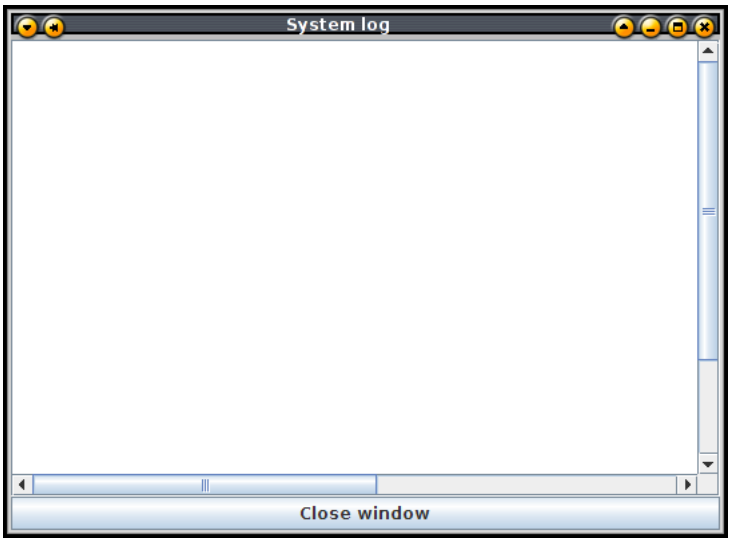

Figure 18: Configuration frame.

UtilFrame and FrameGroup The default implementation of scala.swing is an elegant wrapper on top of Java's Swing GUI toolkit. However, even scala.swing is rather complex, when the development focus is not on GUI. UtilFrame is the base of all GUI frame classes in the simulation framework.

FrameGroup is a utility class for managing groups of windows. It allows closing multiple closely coupled windows with a single mouse click and grouping them to maximize visibility and the ease of use.

6 Domain specific framework layers

Re-implementing various common building blocks in each new CPU design rather quickly becomes counter-productive. The simulator framework comes with a minimal set of generic building blocks available for typical CPU designs. The components and their usage are described here.

6.1 CPU layer

The CPU layer provides minimal functionality for implementing new CPUs. The following features are provided:

- Abstractions for defining CPUs (`CPU, CPUComponent`), instruction sets (`InstructionSet`), instructions (`Instruction`), and executable programs (`Program`).
- Endianess-aware instruction decoders (`InstructionDecoder`) and encoders (`InstructionEncoder`) - with integrated support for reading from and writing to files. The usage of these classes is shown in Algorithm 20 in Section 6.4.3.
- Components for registers (`Register`) and register files (`RegisterFile`).
- A dummy CPU (`DummyCPU`) component for testing.

6.2 Memory layer

The memory layer provides minimal functionality for implementing and using memory and other storage components. The layer mainly provides components for defining generic memory modules, buffers, and caches. A generic constant latency memory module (`SimpleSignalingMemory`, see Algorithm 15) and a generic N-way set-associative cache component (`CacheNWaySetAss`, see Algorithm 16) are also provided.

The word signaling in memory interfaces denotes the ability to signal the depending component when the data arrives if it is not immediately available. If `fetchLatency` is greater than `signalingLatency`, a signal is returned after `signalingLatency` time units. The actual requested data is returned after `fetchLatency` time units. In case of caches, the fetch latency also depends on the source component's latencies.

The memory module can be often used for modeling a wide range of physical memory modules. As can be seen from its constructor interface in Algorithm 15, it can be parametrized with a capacity, address bus width, maximum block size for transfers, concurrent read/write counts and latencies for signaling and fetching of data.

Algorithm 15 SimpleSignalingMemory constructor interface

```
class SimpleSignalingMemory(
    capacity: Int,
    name: String,
    addressBusWidth: Int,
    maxBlockSize: Int,
    concurrentReadCount: Int,
    concurrentWriteCount: Int,
    concurrentReadWriteCount: Int,
    signalingLatency: Int,
    fetchLatency: Int,
    maxCommandCount: Int
) extends SignalingMemoryComponent
```

The cache component has somewhat similar constructor interface. Many of the parameters are the same as above, but there are cache related parameters such as the ones for defining cache, row, and set sizes. The cache can be set up as being either fully associative, direct mapped, or n-way set-associative. The source can be any signaling memory component.

Algorithm 16 N-way set associative cache constructor interface

```
class CacheNWaySetAss(
    fetchLatency: Int,
    signalingLatency: Int,
    cacheSize: Int, // in bits
    rowSize: Int, // in bits
    setSize: Int, // in bits
    maxBlockSize: Int,
    concurrentReadCount: Int,
    concurrentWriteCount: Int,
    concurrentReadWriteCount: Int,
    source: SignalingMemoryComponent
) extends SignalingMemoryComponent
```

6.3 Network layer

The network layer has abstract black-box definitions for various kinds of message sending networks between generic nodes. In SMASim a network abstraction is used for communication between computational nodes. This network consist of a `Bus` object and one or more `Node` objects. An extension to generic nodes, a `Buffer`, is provided. The `Buffer` has an internal queue for incoming messages. As an example, also a star-shaped network bus is provided in the `StarBus` class.

In star bus, all message transmissions have a constant sized latency, to model expected latency of some specific network topology. The bus supports both unicast and broadcast functionality.

6.4 MIPS32 layer

In addition to generic layers listed above, a layer mostly consisting of definitions for the MIPS32® instruction set abstractions and instructions is provided. The supported standard instructions are:

- ADDI, ADDIU, ADD, ADDU, ANDI, AND, BEQ, BGEZAL, BGEZ, BGTZ, BLEZ, BLTZAL, BLTZ, BNE, CLO, CLZ, DIV, DIVU, JALR, JAL, JR, J, LB, LBU, LH, LHU, LUI, LWL, LWR, LW, MADD, MADDU, MFHI, MFLO, MOVN, MOVZ, MSUB, MSUBU, MTHI, MTLO, MUL, MULT, MULTU, NOP, NOR, ORI, OR, SB, SH, SLL, SLLV, SLTI, SLTIU, SLT, SLTU, SRA, SRAV, SRL, SRLV, SUB, SUBU, SW, XORI, and XOR.

The instructions are implemented precisely to be binary compatible with the original MIPS32® instruction set. However, the ALU instructions do not use signaling since no trap instructions are implemented. Also missing are atomic and floating point instructions.

The MIPS32 layer provides a customizable subsystem for defining MIPS32-like instruction sets. The layer provides means for decoding and encoding instructions belonging to three instruction categories R, I, and J [3]. The functionality in CPU framework layer extends this support for deserializing and serializing data from/to files.

Supporting other RISC instruction sets such as ARM requires only minimal adjustments. Making minor modifications to the instruction set e.gin prototyping phase has been made simple by adding support to common set operations such as union and intersection of two sets. The instruction set connects to the execution pipeline via a loosely coupled and flexible interface (described in Algorithm 17). Currently the interface has been built with only MIPS32 in mind, which might require further abstraction when other instruction sets are added.

6.4.1 Defining instructions

All MIPS32-like instructions are supposed to extend an instruction template similar to what is described in Algorithm 17.

[3]For more information about the MIPS32® architecture, see http://www.mips.com/products/architectures/mips32/.

Algorithm 17 Instruction template

```
object Foo extends InstructionGenerator(new TypeICommand(id) {
  override def loadMemory = (Address(1), 4, _:Int ⇒. . .)
  override def loadRegisters = List(rs ⇒rs, . . ., hi ⇒hi)
  override def execute {
    . . .
  }
  override def storeRegisters = List(rt ⇒rd, . . ., lo ⇒lo)
  override def storeMemory = (Address(2), 4, 65535)
})
```

The methods' execution order in the template follows the temporal dependencies of the actions in the standard RISC-style 5-stage execution pipeline. The interface is rather generic, and even our PRAM-based ([3]) moving threads pipeline ([10]) uses this basic interface listed here.

As a concrete example, the definition of the AND instruction as defined in SMASim is given after the instruction template definition in Algorithm 18.

Algorithm 18 AND instruction definition

```
object AND extends InstructionGenerator(new SpecialCommand((4 << 3) + 4) {
  override def execute = rd := rs & rt
})
```

Method `LoadMemory` returns a memory location, size of the data block, and the action to take when loading data from the main memory. Respectively `storeMemory` returns a memory location, size of the data block and the data to store. Method `loadRegisters` defines the list of registers to load from the register file to the pipeline buffers, and `storeRegisters` stores the buffered register values to register file slots, correspondingly. Method `execute` defines the behavior of the instruction execution, which takes place between the register loads and stores.

The instruction template provides macros for addressing parts of the instruction structure such as the register mnemonics (`rt, rs, rd, hi, lo`) as seen in Algorithms 17 and 18. To prevent errors at compile time, some register slots such as rd are not available for instruction types that do not contain the structure. The macros are available in all methods described in the Algorithm 17. Assignment to register slots can be done with the `:=` syntax.

6.4.2 Defining instruction sets

Defining new instruction sets from predefined instructions is rather straightforward, as described in Algorithm 19. A new set can be constructed from indepen-

dent instructions or set operations − and + can be used to derive more sets of the existing ones.

Algorithm 19 Instruction set definitions

```
// assume we have defined instructions ADD, SUB, MUL, DIV, and FORK

object BasicInstructions extends InstructionSet(ADD, SUB)
object MulDivSet extends InstructionSet(MUL, DIV)

val BasicArithmetics = BasicInstructions + MulDivSet
val SpecialInstructions = BasicArithmetics − DIV + FORK
```

6.4.3 Serializing and deserializing

As an example of instructions deserialization, the simulator ships with a minimalistic simple disassembler, inspired by GNU objdump[4]. Algorithm 20 shows the main steps modulo error handling required to build a similar tool. Additionally the tool also re-encodes the instruction stream and writes it to another file.

Algorithm 20 Instruction serialization and deserialization

```
import core.instructions.InstructionEncoder
import arch.mips32.MIPS32Decoder

val program = MIPS32Decoder load "/tmp/input.bin"
val instructionStream = MIPS32Decoder decode program

for (i ←instructionStream)
  println(i)

val program2 = InstructionEncoder encode instructionStream
InstructionEncoder save ("/tmp/output.bin", program2)
```

[4]Objdump is part of GNU Binutils, see `http://www.gnu.org/software/binutils/`

7 Example architecture

So far the documentation has discussed the low level interfaces of the simulator framework. Since the ability to adapt easily to different kinds of architectures is one of the main features of the simulator, this whole section is dedicated to an example of implementing a complete custom MIPS32 inspired RISC architecture with SMASim. In addition, a monitor is provided for measuring the amount of instructions executed by the architecture during the simulation.

7.1 Architecture description

The example architecture consists of a simple pipelined four instruction 5-stage RISC processor. The processor contains a small register file, the execution pipeline, and is connected to a small main memory module with flat address space. The instructions and data share the same read-write memory space.

A predefined program can be loaded to a fixed memory position. The program counter is initialized with another value when the processor is reset. The last register slot is used as the program counter, the first one always contains zero as in MIPS32.

7.1.1 Instruction set

The architecture has the following instruction set which uses the standard MIPS32 bit encoding:

- ADD r1, r2 -> r3 - a 2's complement addition on registers $r1$ and $r2$, stores the result on register $r3$. No overflow checks are done.
- SUB r1, r2 -> r3 - a 2's complement subtraction on registers $r1$ and $r2$, stores the result on register $r3$. No overflow checks are done.
- LW r1, imm -> r2 - loads a word aligned value from memory address referenced by register $r1$ + the immediate value imm. The value is stored to register $r2$.
- SW r1 -> r2, imm - stores the word in register $r1$ to a word aligned value in memory address referenced by register $r2$ + the immediate value imm.

7.1.2 Execution pipeline

The execution pipeline divides the instruction execution into distinct suboperations. The design was inspired by the classic 5-stage RISC execution pipeline. The duties of each stage are:

1. **Fetch** - fetches the next instruction from program counter's address, next address is calculated in a local pseudo-register.

2. **Decode** - decodes the next instruction and fetches register values.

3. **Execute** - executes the addition or subtraction.

4. **Memory Access** - loads or stores data to/from main memory.

5. **Writeback** - writes back the register values to register file.

7.1.3 Architecture parameters

Some basic parameters of the architecture are described in the Table 1.

Miscellaneous	
Word size	32 bit
Clock rate	100 Hz

Memory component	
Memory capacity	64 kB
Memory read ports	2
Memory write ports	1
Memory access latency	1 cycle

Register file component	
Register file size	32 registers
Register file read ports	2
Register file write ports	1
Register file access latency	1 cycle

Table 1: Example architecture parameters.

The parameters can be later used almost as is when defining the specifications of components.

7.2 Implementation

7.2.1 Instruction set

The architecture can benefit from the existing MIPS32 instruction encoders and decoders. In the example architecture we only need to specify the subset of four instructions (`ADD`, `SUB`, `SW`, and `LW`) and construct a new decoder for the instruction set. The implementation is shown in Algorithm 21.

Algorithm 21 Definitions for instruction set, decoder, and disassembler.

```
val instructionSet = new InstructionSet(ADD, SUB, LW, SW)
val decoder = new MIPS32Decoder(instructionSet)
val disassembler = new MIPS32Disassembler(decoder)
```

7.2.2 Configuration parameters

The ConfigurationFrame component can be used to emulate duck typing style programming. This allows defining very lightweight interfaces for configuration and very simple mapping between the specification and implementation. For the example architecure, we use the configuration code from Algorithm 22.

Algorithm 22 Configuration code.

```
object CPUConfig {
  var wordSize = 4
  var regCount = 32
  var regRPorts = 2
  var regWPorts = 1
  var regLatency = 1
  var startAddress = 1000
}

object MemoryConfig {
  var capacity = 65536
  var maxBlockSize = 4
  var latency = 2
  var RPorts = 2
  var WPorts = 1
}
```

7.2.3 Component definitions

The physical components in the simulation are defined next. The setup consists of a memory module, register file, and the CPU. The CPU internals are further divided into five instruction pipeline parts.

Algorithm 23 High level schema of the simulated architecture.

```
class Simulation(program: Program, val decoder: MIPS32Decoder)
    extends MonitoredSimulation[CPU] {
  val insMonitor = attachController(InstructionMonitor(decoder.instructionSet))
  implicit val clock = insMonitor @> 100 Hz
  def CPUs = List(cpu)

  // Code from Algorithms 24..31 goes here

  cpu loadProgram (Address(loadAddress), program)
  cpu.initialize
}
```

High level schema The high level schema of the simulated architecture is show in Algorithm 23. The code defines a simulator container which encloses the physical components in the simulated system. During the initialization, an instruction monitor is connected to the controller clock. The clock instance used by the components connects to the controller clock via the monitor. The component definitions are followed by another initializer for starting the CPU with a program preloaded in its memory module.

Storage components The given architecture consists of a main memory module with a fixed latency property (2) and a simple register file with unit time latency. Algorithms 24 and 25 define these components respectively. Component configuration is imported from the configuration structures previously presented.

Algorithm 24 Storage components, main memory.

```
val memory = new SimpleSignalingMemory(
    capacity = capacity,
    addressBusWidth = wordSize*8,
    maxBlockSize = wordSize,
    concurrentReadCount = RPorts,
    concurrentWriteCount = WPorts,
    concurrentReadWriteCount = RPorts + WPorts,
    signalingLatency = latency,
    fetchLatency = latency,
    maxCommandCount = 4)
```

A MIPS32 like zero register functionality is provided by the framework in form of a ZeroRegister trait. The register file is extended with the trait.

Algorithm 25 Storage components, register file.

```
val registerFile = new RegisterFile[Int](
    regCount = regCount,
    readPorts = regRPorts,
    writePorts = regWPorts,
    latency = 1,
    maxCommandCount = 3) with ZeroRegister // r0 == 0
```

Execution pipeline The low instruction count, fixed memory and register file access latency, and the lack of branching instructions simplify the pipeline design considerably (informal specification in Section 7.1). For example, control flow can be expressed with the implicit data flow. A compact presentation of the

pipeline is expressed in Algorithms 26, 27, 28, 29, and 30. The implementation is split into five parts, following the practice from Section 7.1.2.

Algorithm 26 Execution pipeline, fetch stage.

```
val pipeline = new BaseComponent(maxCommandCount = 10) {
  private def log(pc:Int, msg: ⇒String) = Logger debug ("PC="+pc+": "+msg)

  def start {
    Logger debug "Starting execution."
    cpu.startAddress ⇒: fetch
  }
  def fetch: Command[Address, Any] = fun { pc: Address ⇒
    log(pc.toInt, "Fetching instruction..")
    ((pc, wordSize) ⇒: memory.read) :⇒ value {
      (_: Option[Data]) map { data ⇒
        (pc + wordSize) ⇒: fetch
        ((pc, data) ⇒: decode) :⇒ execute :⇒ memAccess :⇒ writeBack
      }:Any
    } @@ 0
  } @@ 0 @@ "Fetch"
```

Algorithm 27 Execution pipeline, decode stage.

```
  def decode = fun { (pc: Address, data: Data) ⇒
    log(pc.toInt, "Fetching registers && decoding..")
    val instruction = Decoder decode data
    instruction.pc = pc.toInt
    val loads = instruction.loadRegisters
    val loadCmds = loads map { case (slot, temp) ⇒
      slot.value ⇒: registerFile.read :⇒
      value { instruction.regs(temp.value) = _:Int } @@ 0
    }
    value { instruction } @@ 0 dependsOn loadCmds
  } @@ 0 @@ "Decode"
```

Algorithm 28 Execution pipeline, execution stage.

```
  def execute = value { instruction: MIPS32Instruction ⇒
    log(instruction.pc, "Executing " + instruction.name + "..")
    instruction.execute
    instruction
  } @@ 1 @@ "Execute"
```

Algorithm 29 Execution pipeline, memory access stage.

```
  def memAccess = fun { instruction: MIPS32Instruction ⇒
    log(instruction.pc, "Loading␣&␣storing..")

    instruction.loadMemory map {
      case (addr, size, handler) ⇒
        ((addr, size) ⇒: memory.read) :⇒ value {
          (_: Option[Data]) map { data ⇒handler(data:Int) }
        } @@ 0
    }

    instruction.storeMemory map { case (addr, size, data) ⇒
      (addr, new DataA(data) take size) ⇒: memory.write
    }

    () ⇒: value { instruction } @@ 1
  } @@ 0 @@ "MemoryAccess"
```

Algorithm 30 Execution pipeline, write back stage.

```
  def writeBack = fun { instruction: MIPS32Instruction ⇒
    log(instruction.pc, "Writing␣back..")

    val stores = instruction.storeRegisters

    val storeCommands = stores map { case (temp, slot) ⇒
      (slot.value, instruction.regs(temp.value)) ⇒: registerFile.write
    }

    value {
      log(instruction.pc, "Instruction␣execution␣done.")
    } @@ 0 dependsOn storeCommands
  } @@ 0 @@ "WriteBack"
}
```

CPU component The CPU component has a simple task of controlling the aforementioned components. The definition in Algorithm 31 defines means for initializing the CPU state, starting execution, loading foreign programs and displaying useful state information. A large part of the state information is not used in this simple example, but it can be useful in larger simulations with non-trivial amount of state.

Algorithm 31 CPU component.

```
object cpu extends BaseComponent with CPUComponent {
  def reset = for (i ←0 until registerFile.regCount)
    registerFile(i) = 0

  def initialize {
    reset
    pipeline.start
  }

  // use the default loader since the arch has fixed instruction length
  def loadProgram(address: Address, program: Program) =
    loadProgram(address, program, memory)

  def id = 1
  def ISA = "4 Instructions"
  def wordSize = CPUConfig.wordSize
  def startAddress = Address(CPUConfig.startAddress)

  override def children =
    List(memory, registerFile, pipeline)

  override def state =
    children.map("  "+_.state).mkString("\n")

  override def description = super.description +
      "ID: " + id + "\n" +
      "Memory: " + memory + "\n" +
      "Register file: " + registerFile + "\n"
}
```

7.2.4 Monitors

The implementation provides a simple instruction execution monitor that keeps track of the amount of executed instructions. Executed messages are pattern matched in the overridden traceExecution method of the monitor. The first use for pattern matching is to filter out irrelevant messages from the ones between pipeline components. Pattern matching also allows determining the type of the executed instruction to give better statistics grouped by instruction type.

Each time an instruction has gone through all pipeline stages, an associated counter in a map is increased. The monitor keeps track of the counter status, and the status can be observed via a GUI module defined next. The Algorithm 32 presents the monitor implementation.

Algorithm 32 Instruction execution monitor.

```
case class InstructionMonitor(cclock: ControllerClock)
        extends BaseMonitor(cclock) {
  private val counter = new collection.mutable.LinkedHashMap ++=
    Instructions.instructions map
        { _.gen.name } sortWith { _ < _ } map { (_, 0) }

  override def traceExecution(e: Executable) =
    if (e.name == "WriteBack")
      e match {
        case wb: Command1[MIPS32Instruction,_] ⇒counter(wb.p1.name) += 1
        case _ ⇒
      }

  def name = "Instruction monitor"

  def state = "Executed instruction count:\n" + counter.mkString("\n")
}
```

The GUI code for the instruction monitor is presented in Algorithm 33. The monitor just displays a console frame, the contents of which are updated periodically.

Algorithm 33 GUI code for the instruction execution monitor.

```
class InstructionMonitorGUI(val insMonitor: InstructionMonitor) extends
MonitorFrame("Instruction") {
  def tick = updateText(insMonitor.state)
}
```

7.2.5 GUI

The graphical user interface of the simulator is split into two parts, the main menu and the simulator controller. The main menu (shown in Algorithm 34) can be automatically generated to a great extend. The customized part in this example is the connection between configuration objects described in Algorithm 22. The program loading functionality is automatically enabled when the GUI is told to execute programs.

The simulator controller (shown in Algorithm 35) also comes with a lot of default functionality for controlling the simulation (e.g.reset and step buttons). We only add the code for displaying the instruction monitor and for constructing the customized simulation object. The built-in classes provide routines for easily extending the simulator with various kinds of monitors.

Algorithm 34 Main class for launching the GUI.

```
object SimuGUI extends CPUSimuGUI {
  Logger.debugmode = true

  val versionString = "Example␣architecture␣simulator"
  val executesPrograms = true

  /** Instruction set setup (already shown) */

  val simuControllerFrame = new ProgrammableSimuControllerFrame(SimuGUI)

  val configurationGroup = new ConfigurationFrameGroup(
    simuControllerFrame)

  val CPUConfFrame = new GroupConfigurationFrame(
    "CPU", CPUConfig, configurationGroup)

  val MemoryConfFrame = new GroupConfigurationFrame(
    "Memory", MemoryConfig, configurationGroup)

  lazy val configureButton = Button("Configure␣parameters") {
    configurationGroup.show
    simuControllerFrame.hide
    configurationGroup goTo nextTo
  }
}
```

Algorithm 35 GUI class for controlling the simulation.

```
val simuControllerFrame = new ProgrammableSimuControllerFrame(SimuGUI) {
  type SimuType = Simulation

  def createSimulation = new Simulation(program, decoder)

  def monitors = List[MonitorHandler](
    new MonitorHandler("Show␣instruction␣monitor", {
      val instructionMon =
        new InstructionMonitorGUI(simulation.insMonitor)
      simulation attachTickable instructionMon
      instructionMon
    })
  )
}
```

8 Installation and usage

The simulator is freely available under the GNU Affero General Public License[5] and the distribution package is located at:

- `http://staff.cs.utu.fi/research/MOTH/`

The distribution consists of the complete source code written in Scala (directory `src/`), executable binaries (directory `bin/`), a *nix script for starting simulations (`start.sh`), this report (`manual.pdf`), and the AGPL license text (`COPYING`).

8.1 Software dependencies

The simulator depends on a working installation of Java runtime environment (6 or later) and Scala 2.8 runtime libraries, though as of this writing Scala 2.8 is still considered a beta release. The libraries can be freely downloaded from the following sites:

- Java 6, `http://java.sun.com/javase/6/`
- Scala 2.8, `http://www.scala-lang.org/node/212`

The precompiled simulator depends on Java runtime enviroment (J2RE, tested with the Sun, IBM, and OpenJDK Java distributions) and Scala 2.8 runtime libraries (`scala-library.jar`, `scala-swing.jar`). The default class path settings may need to be reconfigured depending on your setup.

Compiling the simulator also requires the Scala 2.8 compiler, which is part of the standard Scala distribution.

In both cases dependencies contain necessary binaries to run the tasks. The binaries should be accessible via the PATH environment variable.

8.2 Starting the simulation

The simulator can be started by providing a simulation main class (for instance inherited from `CPUSimuGUI`) instance to the aforementioned script `start.sh`. The script implicitly adds current directory to the class path. The script calls the Scala binary, which launches the Java virtual machine with appropriate parameters for launcing the simulation.

For quick testing, the architecture from Section 7 is bundled with the simulator. It can be launched with:

```
$ ./start.sh test.ExampleArchSimulator
```

[5] See `http://www.gnu.org/licenses/agpl.html`

8.3 Using the graphical interface

A typical workflow in using the simulator is to

1. first import an executable from the file system (first button in Figure 19 opens the file selection dialog shown in Figure 20), then to

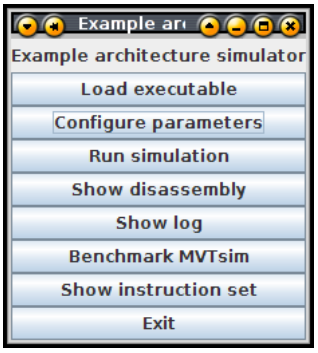

Figure 19: Main menu.

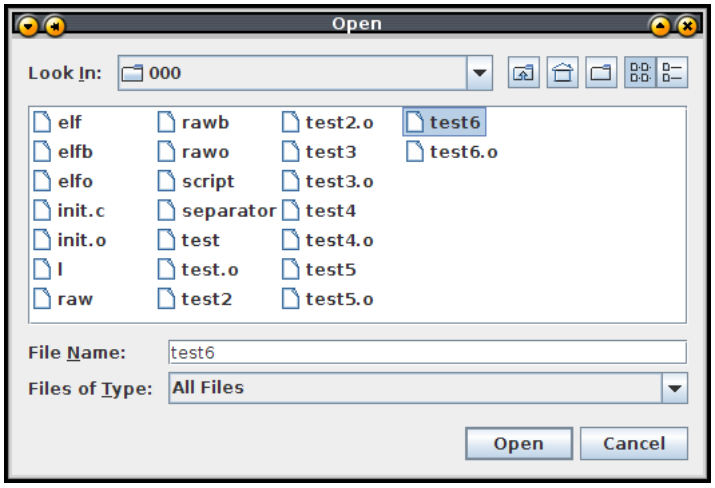

Figure 20: Importing an executable from the file system.

2. configure simulation settings (second button in the Figure 19), and

3. start a simulation (third button in the Figure 19) with

4. appropriate monitor frames opened from the controller menu (Figure 21).

Figure 21: Simulation controller menu.

The simulation restarts whenever the simulation settings are reconfigured, a new executable is imported to the simulator, or the reset button is pressed on the controller menu.

The simulator does not support any undo functionality at the moment and the simulation can only be run forward, stepwise in groups of 1, 10, 100, 1000, or 10000 cycles. The code behind the default user interface can easily be adjusted to support other kind of execution modes or e.g. remote controlling via a network.

In the currently available version, configuration settings cannot be yet saved to a file and the log output also has to be manually copied from the standard output or from the logger frame. A future version will fix these issues.

References

[1] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, pages 59–67, 2002.

[2] Sangyeun Cho, Socrates Demetriades, Shayne Evans, Lei Jin, Hyunjin Lee, Kiyeon Lee, and Michael Moeng. Tpts: A novel framework for very fast manycore processor architecture simulation. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 446–453, Washington, DC, USA, 2008. IEEE Computer Society.

[3] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[5] Joseph JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[6] J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, and S.Y. Han. FaCSim: a fast and cycle-accurate architecture simulator for embedded systems. *ACM SIGPLAN Notices*, 43(7):89–100, 2008.

[7] D. Mihocka and S. Shwartsman. Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure.

[8] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[9] PM Ortego and P. Sack. SESC: SuperESCalar Simulator, February 2007.

[10] J. Paakkulainen, J-M Mäkelä, V. Leppänen, and M. Forsell. Outline of risc-based core for multiprocessor on chip architecture supporting moving threads. In *CompSysTech '09: Proceedings of the International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing*, pages 1–6, New York, NY, USA, 2009. ACM.

[11] MT Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. Performance Analysis of Systems and Software, 2007. ISPASS 2007. In *IEEE International Symposium on*, pages 23–34, 2007.

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Information Technologies

Turku School of Economics
- Institute of Information Systems Sciences

ISBN 978-952-12-2425-6
ISSN 1239-1891