

SyncSim Extensions

Simulation with VHDL and Code Generation

Jan Enoksson
Simon Olsson

Luleå University of Technology
MSc Programmes in Engineering
Computer Science and Engineering
Department of Computer Science and Electrical Engineering
EISLAB

SyncSim extensions:
simulation with VHDL and code generation

Jan Enoksson
janeno-1@student.ltu.se

Simon Olsson
simols-1@student.ltu.se

January 20, 2006

Abstract

SyncSim is a simulator framework capable of loading different simulator modules. It is used today with a module that simulates hardware models described with Java. This simulator module is used together with a model of a MIPS processor core in courses given at EISLAB. The purpose of this thesis is to create a new simulator module for SyncSim which can use hardware models described with a mixture of VHDL and Java and to implement a C compiler that can generate code compatible with the existing MIPS model.

EESim is a simulator module for an early version of SyncSim that uses VHDL to describe the hardware model. This simulator module will be extended to meet the requirements of the new simulator. The Portable C Compiler (PCC) has previously been released as open source and work has been done by others to modernize it. PCC will be ported to the MIPS architecture and its portability evaluated.

The result is a new version of EESim that can simulate models which mix VHDL and Java, and a PCC port capable of producing MIPS assembly code. With regards to portability PCC was found to be relatively easy to modify for use on a new computer architecture.

Preface

This thesis is the result of our final semester of the Master of Science in Computer Science and Engineering program. The work was done during the fall and winter of 2005 at the Embedded Internet Systems Laboratory (EISLAB) at the Department of Computer Science and Electrical Engineering (CSEE) at Luleå University of Technology.

We would like to take this opportunity to thank a couple of people who have made this thesis possible. Firstly, we would like to thank Per Lindgren for his role as the supervisor of this thesis. We would also like to thank Anders Magnusson for taking time out of his busy schedule to answer our questions, and for his enthusiasm for PCC.

Contents

1	Introduction	1
1.1	Background	1
1.2	Purpose	1
1.3	Limitations	2
2	EESim	3
2.1	Background	3
2.1.1	Hardware description languages	3
2.1.2	SyncSim	3
2.1.3	EESim	4
2.2	Implementation	7
2.2.1	Summary of changes	8
2.2.2	The NCSim interface	8
2.2.3	Interfacing with SyncSim	9
2.2.4	Co-simulating Java and VHDL components	10
2.2.5	The design XML-file	11
2.3	Evaluation	12
2.3.1	Code layout	12
2.3.2	Rewinding	12
2.3.3	NCSim synchronization	13
2.3.4	Design XML-file changes	13
2.3.5	EESim dependencies	13
2.3.6	Test case - running a mixed VHDL/Java MIPS design	13
3	PCC	15
3.1	Background	15
3.1.1	The C language	15
3.1.2	History of PCC	15
3.1.3	PCC concepts	16
3.1.4	The MIPS architecture	18
3.1.5	Endianness	19
3.2	Implementation	20
3.2.1	Endianness in the MIPS port	20
3.2.2	Translation table	20
3.2.3	Calling convention	23
3.2.4	MIPS assembler	24
3.3	Evaluation and results	25
3.3.1	Missing data type support	25
3.3.2	Optimization	25
3.3.3	The assembler	25
3.3.4	Porting	26

4	Conclusions	27
A	EESim Manual	29
A.1	Introduction	29
A.2	About EESim	29
A.2.1	When to use EESim	29
A.2.2	Limitations with simulation in EESim	30
A.3	Creating your own design	30
A.3.1	The XML file	30
B	EESim class diagram	32
C	PCC file structure	33
D	Fibonacci assembly code	34

Abbreviation list

ALU	Arithmetic and Logic Unit
ANSI	American National Standards Institute
CAD	Computer Aided Design
CFC	C Function Call library
CISC	Complex Instruction Set Computer
CVS	Concurrent Versions System
ELF	Executable and Linking Format
GCC	GNU C Compiler
GUI	Graphical User Interface
HDL	Hardware description language
ISA	Instruction Set Architecture
JNI	Java Native Interface
JVM	Java Virtual Machine
MAS	MIPS Assembler
PCC	Portable C Compiler
RISC	Reduced Instruction Set Computer
VDA	VHDL Design Access library
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Background

Today an application called SyncSim [7] is used in courses given by Embedded Internet Systems Laboratory (EISLAB) at Luleå University of Technology (LTU) to simulate synchronous digital designs. It is a Java based simulator framework that lets the user load designs of different hardware models. Different simulator modules can be loaded into SyncSim depending on what to simulate and how the model is described. This means that SyncSim is very versatile and simulator modules can be designed for specific purposes. For example, SoftSim is one implemented simulator module that uses Java to describe hardware. It is the default simulator module that comes with SyncSim.

SyncSim provides the Graphical User Interface (GUI) that lets the user control the simulation, and it also displays the visual representation of the simulated model. The user can invoke actions such as stepping forward and backward in the simulation and can view the states of different components at the current cycle of the simulation.

A design in SyncSim consists of a number of Java classes that describe the different parts of the design. These classes define both a part's function and also its graphical representation. A XML-file is used to specify how the parts are connected and their placement in the graphical view of SyncSim.

There exist a model of the MIPS computer architecture for the SoftSim simulator (figure 1.1 shows SyncSim running SoftSim with a model of a MIPS [1] processor core loaded). This model among others, together with SoftSim, is used for educational purposes at LTU.

1.2 Purpose

SoftSim lets the user work with a higher degree of abstraction of the hardware than current Hardware Description Languages (HDLs), and could possibly in the future allow even more abstract models; however, when describing hardware that is going to be realized (in silicon or programmable logic) a HDL is needed. To make the process of translating an abstract SoftSim model into a design realizable in hardware, and to make it possible to have parts of a design written in a HDL (for convenience or by preference) a new simulator module for the SyncSim framework is needed.

To run program code on the MIPS model for SoftSim a set of compiler tools is needed. It should be possible to compile both C and assembly code for the MIPS. These tools need to, preferably, be public domain as well as relatively small and easy to modify.

So, the purpose of this thesis is two-folded. Firstly, to enable SyncSim to simulate designs with parts described in a HDL, by creating a new simulator module. Secondly, to implement easily modifiable compiler tools for use with the MIPS model.

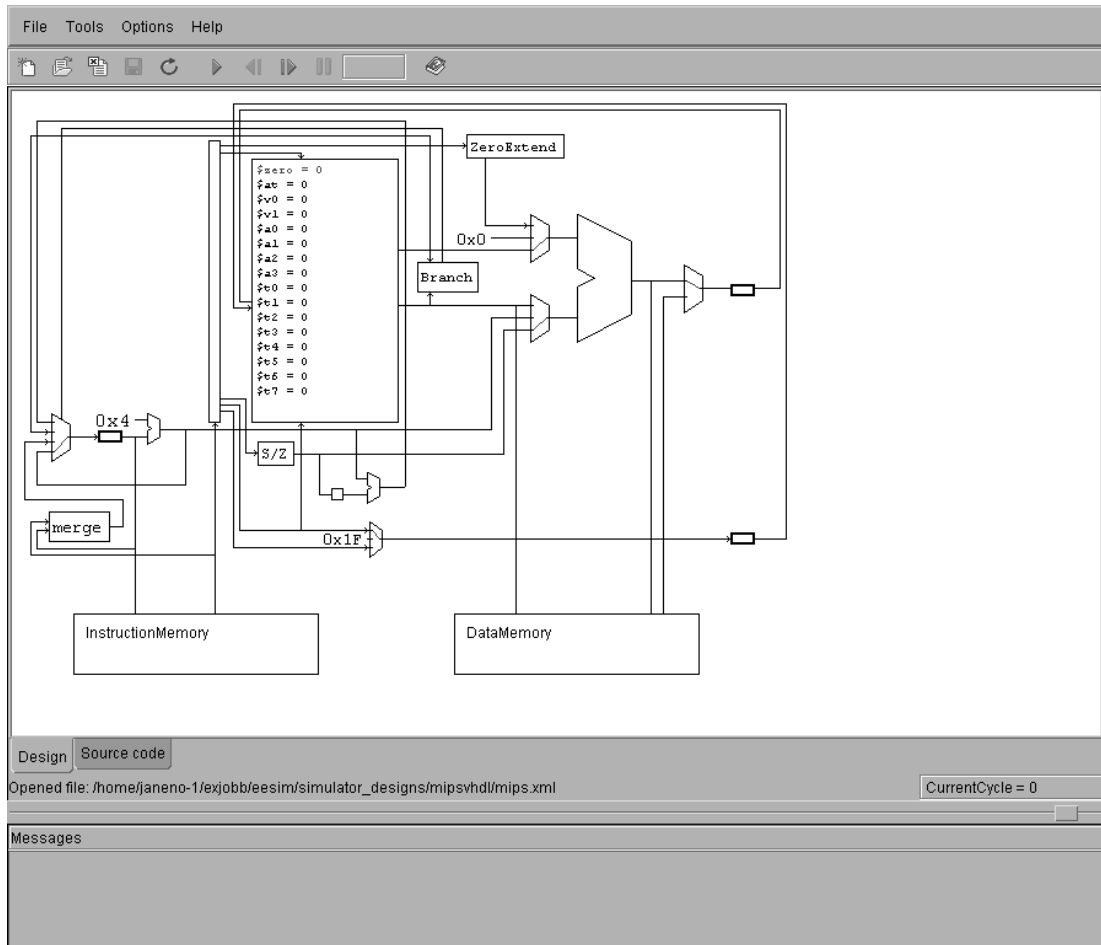


Figure 1.1: SyncSim screenshot

1.3 Limitations

We will use an existing simulator for HDLs and create an interface that a SyncSim simulator module can use when some parts of the design are described by a HDL.

EESim [8] is an already existing simulator module for an early version of SyncSim that can simulate a MIPS [1] processor model written in VHDL (a hardware description language). It implements an interface towards a HDL simulator, Cadence NCSim [6], to accomplish this. NCSim is capable of simulating a few different HDLs, but this thesis will be limited to models described in VHDL.

The focus for the creation of the compiler tools will be on code generation, that is, compiling C code into assembly code.

The Portable C Compiler (PCC) [4] is a relatively small compiler that is considered to be relatively easy to port to new platforms. It can compile code that suits our purposes and will be evaluated as a compiler to be used in conjunction with the MIPS model.

Chapter 2

EESim

2.1 Background

This section will introduce concepts and software that we have used in our work with EESim.

2.1.1 Hardware description languages

HDLs are used to write specifications of hardware that can be used in different stages of hardware development. Simulation, testing and synthesis are some tasks that can be performed on a hardware model written in a HDL. There are many different HDLs used today but one of the most well known is VHDL. It is VHDL that we have used in our work.

One type of CAD tool used with HDLs are simulators. We have used a simulator from Cadence called NCSim [6]. NCSim can make use of something called test-benches to test a design. These test-benches can contain, for example, a pre-defined set of tasks to perform as well as the expected outcome. A comparison between the expected outcome, or result, can then be compared to the actual result that the design produces. Test-benches also, typically, control the clock- and reset-signals for a design.

2.1.2 SyncSim

Originally SyncSim [7] was developed as a student project at Luleå University of Technology in the spring of 2004. For a long time the MIPS simulator SOLL had been used in the introduction course to computer engineering. The idea of SyncSim was to replace this simulator with a more general simulator that could be used to simulate a model of the MIPS as well as be capable of simulating any synchronous design. Figure 2.1 shows a screenshot of the MIPS model loaded into SyncSim and figure 2.2 shows windows displaying the contents of the code and data memory of the MIPS model. The code loaded into the MIPS model is a program that calculates some numbers of the fibonacci series and can be viewed in section 3.1.1. The view of the data memory contains the calculated fibonacci numbers and the code view shows the program loop where the numbers are calculated with current instruction highlighted.

Since the end of the student project that initiated the development of SyncSim a new version has been released. SyncSim has replaced SOLL and is now used in two of the courses at the department.

SyncSim itself is not a simulator but a framework for synchronous simulators. For a simulator to run within SyncSim it must follow some rules, that is to extend SyncSim classes. All designs that are loaded by SyncSim are defined by a XML-file. This file includes all objects and their attributes of the design and describes how all objects are connected. This is independent of what simulator is used. The simulator to be used with the design is specified in the XML-file.

The simulator that is distributed with SyncSim is called SoftSim. This simulator uses only Java classes to describes a design and no other resources are used. Because no system dependent

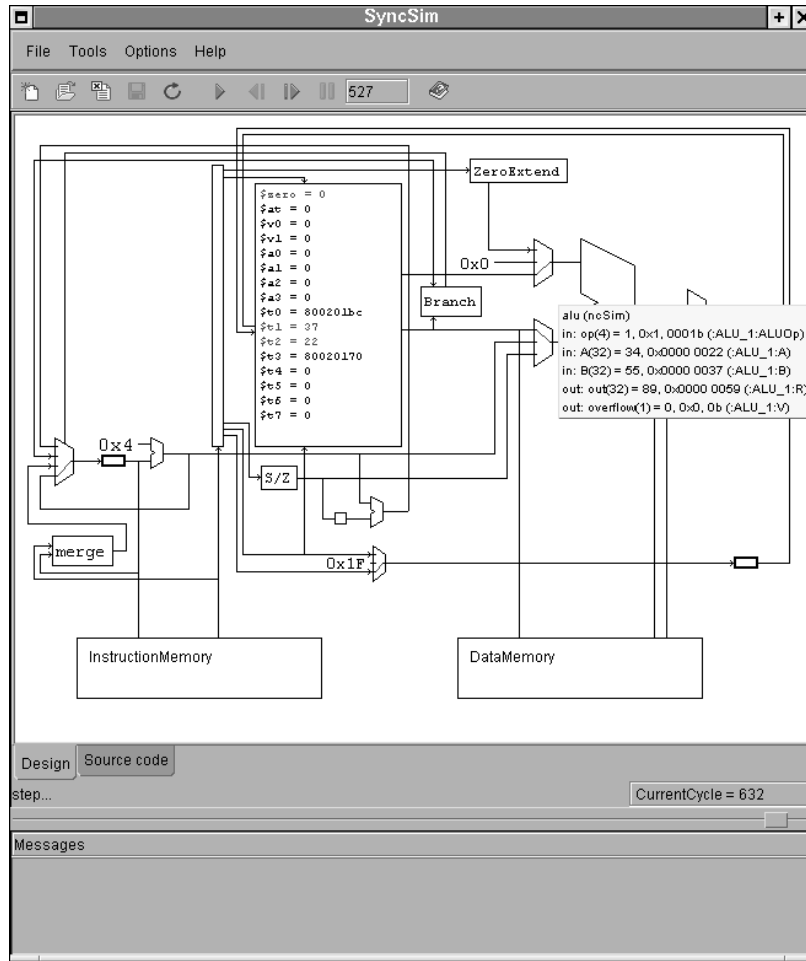


Figure 2.1: SyncSim screenshot with tooltip

functionality is used this simulator can easily be moved between different systems as long as the Java Virtual Machine (JVM) is installed. As we will see later this is not the case with EESim.

SoftSim has two different classes that parts in the design can inherit from: *SoftPart* and *SoftSyncPart*. The first is used by asynchronous parts and the second by synchronous parts. Both parts contain an *update()* and *step()* method, but only the *update()* method is implemented in *SoftParts*. The *step()* method is what allows *SoftSyncParts* to act in a synchronous manner.

2.1.3 EESim

At the same time as the first version of SyncSim and SoftSim was developed another student project worked on a second simulator module called EESim [8]. The idea of this simulator was to visualize the simulation of a model of a MIPS [1] described in VHDL. To do this Cadence NCSim [6] was used to simulate the model. All parts of the simulation took place in NCSim and EESim was actually just an interface for communication with NCSim.

When starting NCSim a C-library file can be loaded. When running NCSim with EESim this library file contains functions used for communication with EESim (figure 2.3).

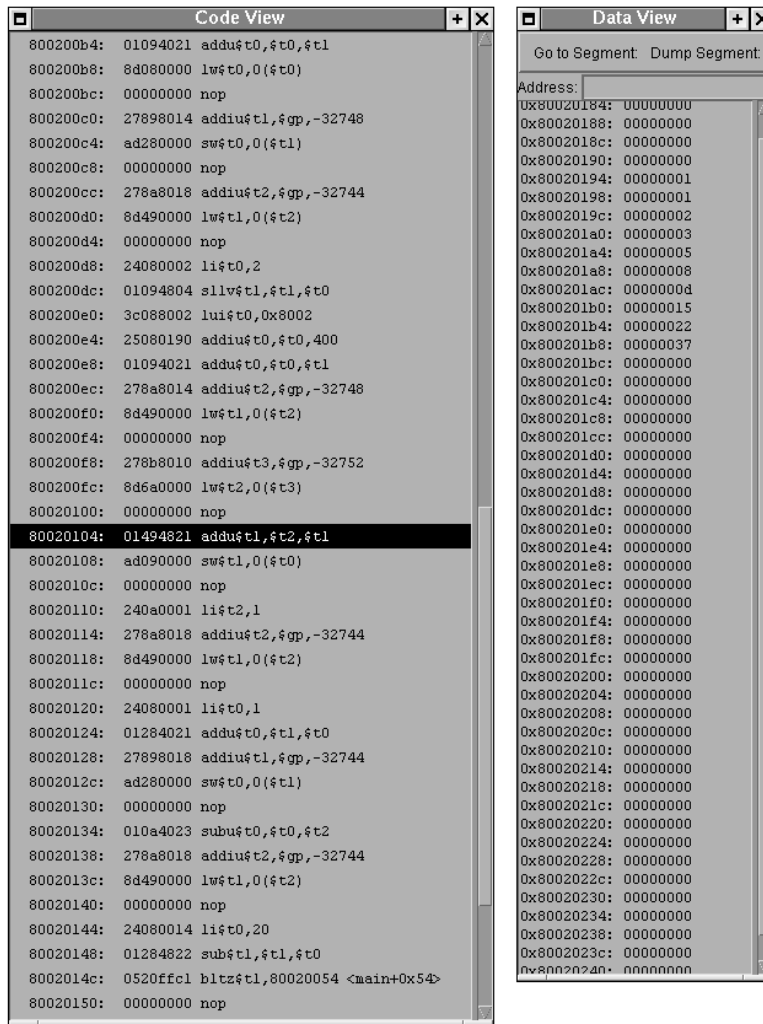


Figure 2.2: SyncSim's data and code view windows

Reading signals and controlling the simulation

Because the whole simulation took place in NCSim, signal values were only needed to be sent in one direction: from NCSim to EESim. EESim used one Java class for each signal that should be visualized. The class that represented a signal had to know the name and type of the corresponding signal in NCSim. This information was hard coded in the classes.

To control the simulation in NCSim communication was sent in the opposite direction. Forwarding one clock cycle was done by sending a command, same as the one used when running NCSim in normal mode interacting directly with the program, to NCSim telling it to run for the number of seconds corresponding to one clock cycle. The length of a clock cycle was specified as the frequency in the XML-file of the loaded design. Clock signal edges was generated by a test-bench that was run with the design in NCSim. This means that the clock frequency was specified at two different places, in the XML-file and in the test-bench, but at the same time SyncSim does not care about the length of a clock cycle.

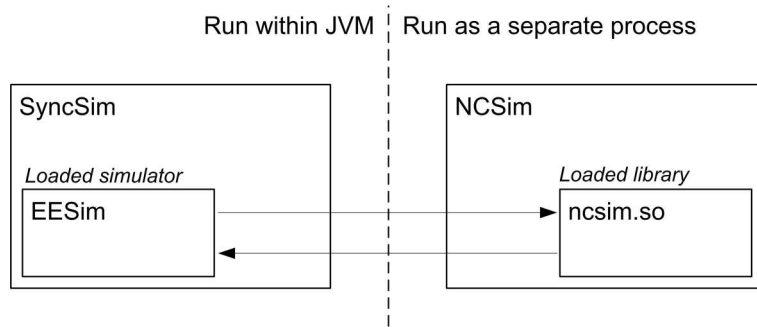


Figure 2.3: Relation between SyncSim, EESim and NCSim

Communication

For the communication between EESim and NCSim two POSIX [11] message queues are used. One message queue for sending control signals from EESim and the other for acknowledgements and returning signal values. Messages sent over a message queue are asynchronous but for the communication between EESim and NCSim two messages are always sent. The communication is always initiated by a message from EESim and a reply is always sent from NCSim. The return message can either be an acknowledgement or a value for a requested signal.

EESim uses two threads and NCSim one thread for the communication over the message queues (see figure 2.4). The first thread in EESim, the simulator thread, takes an action posted by the user via SyncSim's GUI and executes it. Each action will send a number of, depending on the type of the action and the size of the design, messages to NCSim. When the simulator thread has posted a message it goes to sleep. For actions that sends more than one message to NCSim the second message will not be sent until a reply of the first message has been received.

NCSim's thread, called communication thread, listens for messages from EESim. The communication thread will block on the receive function until the first message arrives. The message will be processed and a return message will be sent back over the other message queue to EESim. This message can be a return value or just an acknowledgment. After the message has been sent this thread will call the receive function again and receive a message that is waiting or block until a new message arrives.

The message sent from NCSim will be received by the second thread in EESim, the return thread. The return thread has been blocked on a receive function until the first message has been received. The received return value from NCSim is stored and the simulator thread is notified. The return thread will go back and block on the receive function and the simulator thread can, after it has processed the return value, post a new message to NCSim. According to the principles stated above and that the simulator thread is the only thread posting messages to NCSim it is certain that only one message is present in the two queues at any one time.

EESim uses two C libraries to be able to access data from NCSim: the C Function Call library (CFC) and the VHDL Design Access library (VDA). CFC allows C functions to be called from NCSim's built-in command language. These C functions then uses VDA to access objects in the VHDL model.

NCSim allows users to link C code into the simulator at start up. This C code is compiled in the form of a shared library file. The file contains code to communicate via the message queues and to set and get values of signals in the running simulation. Functions in the shared library file uses CFC and VDA for these purposes. CFC is used so that NCSim can call a function containing an infinite loop that listens for messages on the message queues.

See the product documentation for NCSim [6] for further details on VDA and CFC.

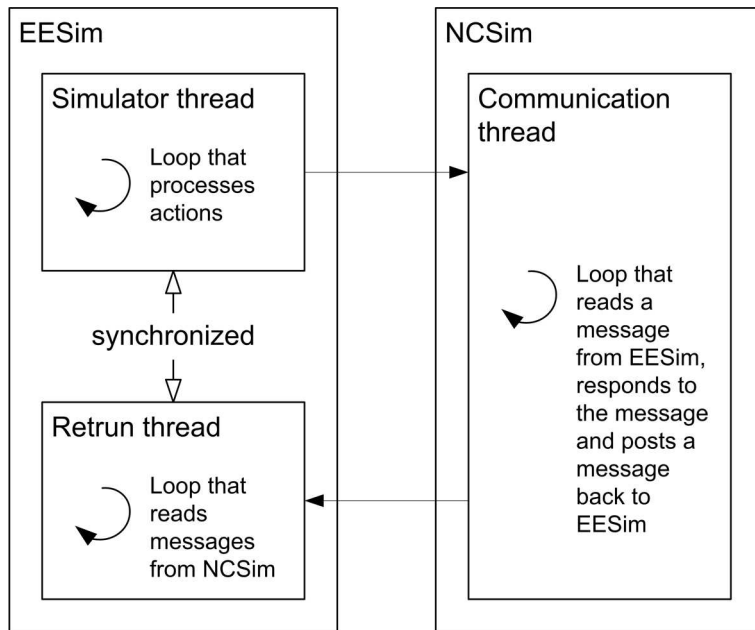


Figure 2.4: Threads used by EESim for communication with NCSim.

The Java/C interface

Posting messages to and reading messages from message queues are done with system calls in C. SyncSim is written in Java and simulators that are loaded are Java classes. To connect the C-code needed by EESim for communication over the message queues the Java Native Interface (JNI) [13] was used. This C-code should not be confused with the shared library that is loaded into NCSim. They are two completely different files.

Data types

VHDL has different signal types. For each type special functionality is needed to send that kind of signal from NCSim to EESim. Objects in EESim that represented signals in NCSim called different method for fetching signals depending on that signal's type. For example, the method *getIntegerValue()* was called to fetch a VHDL signal of the integer type.

Bit width limitations

When modeling a 32 bit MIPS architecture (which is what EESim did) there is no need to be able to send any signals with larger bit widths. Because of this there were some bit width limitations in the first version of EESim. Most of these limitation where found in methods for radix conversions. For a simulator that should run designs with arbitrary bit width all these kinds of limitations must be eliminated. SyncSim uses the Java standard API class *BigInteger*, which has no limitation in the size of a number it can represent, for signals.

2.2 Implementation

In this section the changes that was made to the original version of EESim will be described and explained. No examples of actual code will be presented and the interested reader is referred to the CVS repository [7] containing both the source code for SyncSim as well as EESim. Javadoc documentation for EESim can be generated from the source code and may be consulted during the

perusal of this section if more details of certain classes is sought. There also exist a class diagram for EESim, in Appendix B, that may help clarify some aspects of the program.

2.2.1 Summary of changes

To change EESim from a special purpose MIPS simulator to a general purpose synchronous simulator the following points need to be addressed:

- Make EESim compatible with SyncSim 2.0
- Make it possible to co-simulate parts written in both Java and VHDL
- Add support in EESim for setting signals in NCSim
- Change so that EESim controls the simulation instead of letting NCSim rely on a test-bench
- Add standard Java classes for parts that are simulated by NCSim
- The user should not need to specify type and size, just the name, of a VHDL signal in a design for EESim
- Remove the concept of a clock cycle's length from EESim
- Implement synchronization that lets more than one thread post messages to NCSim
- Make the installation of EESim easier; remove hard coded paths
- Minimize the paths specified in the XML-file of a design
- Remove all dependencies on the MIPS model from EESim
- Remove all limitations in bit width

Each point will be elaborated later in the document. Some points are not strictly necessary for functionality, but were implemented to make usage of EESim easier.

2.2.2 The NCSim interface

Shared library modifications

The existing interface needed to be modified so that it is possible to set the values of signals in NCSim, to enable EESim to mix VHDL and Java components in a design (see section 2.2.4 for more information about this). Since signals in NCSim can have different data types some kind of handling of this had to be done. To make the work of potential users easier it was decided that EESim should handle this and remove that task from the users. This meant that the code for the shared C library had to be extended with functions that could determine the size and type of a signal. These functions also guarantees that the user never tries to access signals with faulty sizes or types, which makes the design process a bit less error prone.

Communication between NCSim and EESim was already handled in a satisfactory manner. The only modifications, to the shared library file, was to implement a new kind of message that caused NCSim to exit gracefully, that is, to shutdown in the same manner as if it had been exited manually and to add messages that can set values of signals in NCSim.

Structural changes

EESim's strong connection to the simulation of a MIPS model had to be removed in order to make EESim a more general simulator, capable of simulating any kind of synchronous circuits. This was easily done, since it only meant removing methods for handling MIPS specific data and compilation of MIPS code.

Previously, EESim had a public statically declared instance of the *NCSimulator* class (which implements methods for communicating with NCSim), which allowed *EESimulator* (which is the simulator module class that is loaded into SyncSim) and *Nc(Sync)Part* to access methods for the interface to NCSim. Another way of doing this is to write the *NCSimulator* class as a Singleton [5]. That is how *NCSimulator* is implemented now.

Only one process of NCSim is run and used to simulate all different parts of the design that is described by VHDL, hence we only want one instance of the *NCSimulator* class that represents NCSim. Each class that represents a piece of the loaded design that is simulated by NCSim needs to access the *NCSimulator* class directly to set and fetch signal values. Both these criterias makes a Singleton a good choice for implementing the *NCSimulator* class.

A stronger synchronization had to be added to make it possible to have different threads post messages to NCSim, whereas the previous version of EESim relied on that only one thread did this. A second thread, beside the simulator thread, may need to know the value of a signal in NCSim, for example if the thread needs the value of the signal to be able to show the graphical representation of the design in SyncSim. To make it possible for more than one thread to use the same pair of messages queues we have assigned an unique id to each message. When a thread that is waiting on a reply from NCSim is notified from the return thread it checks if a return value is available and if it has the right message id. Only if both these criteria's are meet will it continue to process the returned value. If the returned message don't have the right id the thread will go back to sleep and wait for the next message from NCSim.

Instead of relying upon a test-bench written in VHDL to control clock and reset signals in NCSim, EESim now explicitly sets these signals as needed (see sections 2.2.4 and 2.3.3).

Processes executed from the Java Virtual Machine (JVM) uses internal buffers in the JVM to store its output streams (standard output and standard error). If an executed process generates to much output without emptying these buffers, it will cause the process to block while waiting for the buffers to be emptied. The executed process in EESim's case is the NCSim simulator. To solve this problem EESim has two threads running which each reads from the standard output stream and the standard error stream. This keeps the buffers from filling and NCSim can be run indefinitely.

The previous version of EESim had several paths hard coded in the source code, to locate various resources. All of these paths have now been removed and the XML-file properties (see section 2.2.5) and the environment variable (see section 2.2.3) replaces them.

2.2.3 Interfacing with SyncSim

Development of EESim's interface to NCSim and the SyncSim simulator module was developed seperately from the beginning. The NCSim interface was first tested and debugged using the old SyncSim version. Only after the additional features had been added and their functionality verified was compatibility with the new version of SyncSim implemented. This was done to ease the transition and separate it from the process of rewriting the interface to NCSim.

To modify EESim for use with the new SyncSim, the *EESimulator* class had to be rewritten. Since the source code for SoftSim was available and working with SyncSim, the decision was made to use SoftSim's code as a base and modify that to comply with the requirements of EESim. This decision led to a relatively simple transition.

The changes that had to be made, to the SoftSim code, were all minor additions or alterations of the code. The *step()* method of the *StepAction* class in *EESimulator*, as documented in section 2.2.4, was the most important change, since it allowed synchronization between NCSim and EESim.

An environment variable was introduced to specify the directory where EESim resides. Functionality to handle this variable as well as the new XML-file properties (see section 2.2.5) was added to *setupComplete()* in *EESimulator* to be used when initializing the interface to NCSim.

The new version of SyncSim can keep track of how many processor cycles has been run, so this functionality was removed from EESim.

2.2.4 Co-simulating Java and VHDL components

The difference between simulating a design entirely written in VHDL in EESim and simulating a mixed Java and VHDL design is that an interface is needed between components written in Java and components written in VHDL. There must be a way that signals from a Java component gets passed on to the VHDL part being simulated in NCSim and vice versa. This is done by introducing two classes, *NcPart* and *NcSyncPart*, that encapsulates an asynchronous respectively synchronous VHDL component. The two parts uses the interface implemented in the *NCSimulator* class to communicate with NCSim and access the values of the VHDL component they encapsulate.

These two parts are used in the same manner — from the user’s perspective — as ordinary Java component written for the SoftSim simulator, except for some changes to their XML entries (see section 2.2.5). By encapsulating the VHDL components in this way, we can let all communication between EESim and NCSim be handled by instances of *NcPart* and *NcSyncPart*. So whether a component is written in Java or VHDL will not matter to the simulator. This means that the simulator for combined VHDL and Java components can be based upon and be very similar to the existing SoftSim simulator. It also relieves the user of having to write any Java code to incorporate a VHDL component into a design.

The existing interface that implements passing of messages between NCSim and EESim can be kept and extended so that it includes support for setting values of signals in NCSim. *NcPart* and *NcSyncPart* utilizes this interface as figure 2.5 demonstrates.

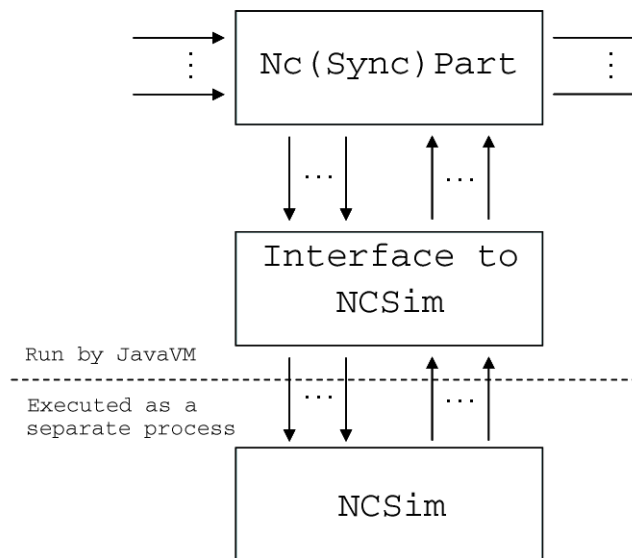


Figure 2.5: Communication between EESim and NCSim

Simulating a step forward to the next processor cycle in the Java simulator needs to trigger an update of signal values in the VHDL components. This means that NCSim somehow must be made aware that it is time to simulate all components, so that the out-signals of all VHDL components are the result of the current in-signals propagated through the logic of each component. *NcPart* and *NcSyncPart* will behave a bit differently from ordinary Java parts because of how synchronicity is implemented in the Java simulator.

Asynchronous parts in the Java simulator have an empty *step()* method and an *update()* method which does calculations on the current in-signals and sets the corresponding values on the out-ports. *NcPart*'s *update()* method does, in addition to this, also send the in-signals to NCSim and runs NCSim's simulation forward to propagate the value through the design. After which the *NcPart* requests the out-signal values from NCSim and copies these values to the corresponding out-ports (see table 2.1).

Table 2.1: Calling the update method of a NcPart

- | |
|--|
| <ol style="list-style-type: none"> 1. <i>update()</i> is called. 2. Send values of in-signals to NCSim. 3. Run NCSim forward to propagate signals. 4. Fetch values of out-signals from NCSim and copy them to the out-ports. |
|--|

Synchronous parts in the Java simulator usually just copies the values of their in-ports as a first stage and then makes its calculations before, as a second stage, the current value of the relevant signals are copied to the components' out-ports. The first stage is completed in the *step()* method of a part and the second stage is done in a part's *update()* method. For *NcSyncPart*, which is synchronous, the *step()* method sends the in-port values to NCSim and the *update()* method copies the values of the out-signals in NCSim to the out-ports of the *NcSyncPart*.

Synchronous components simulated in NCSim must get the values from the first stage and then propagate these signals through their logic, so that all internal states and signals are updated. Only after this is done can the second stage come to pass without any unwanted side effects.

To accomplish this coordination EESim generates a rising edge (see section 2.3.3 for a more detailed discussion of this) on the clock signal in between the two above mentioned stages. This triggers an update of the registers in all VHDL components in NCSim and these values are then propagated by running the NCSim simulation forward, (see table 2.2).

Table 2.2: Updating synchronous parts (NcSyncPart)

- | |
|--|
| <ol style="list-style-type: none"> 1. <i>step()</i> is called on all parts. <i>NcSyncParts</i> send values of in-signals to NCSim. 2. Generate rising edge on clock signal. 3. Run NCSim forward to propagate signals. 4. Run the <i>update()</i> method for all parts. <i>NcSyncParts</i> fetches values from NCSim and copies them to the out-ports. |
|--|

The above process is repeated for every invocation of the *step()* method in the *StepAction* class in *EESimulator*.

2.2.5 The design XML-file

There are some new properties for the XML-file, describing the design, compared to *SoftSim*'s XML-file. All of these properties have to do with NCSim and VHDL related matters. The manual for EESim can be found as Appendix A and details the new properties.

The goal of the new XML properties is to make it possible to create a design with VHDL components without having to write any Java code. This is accomplished by giving instances of *NcPart* and *NcSyncPart* the *ncIn* and *ncOut* properties, which contains the names of the VHDL signals. These names are connected to the in- and out-ports of the Java part by matching them to the corresponding entry in the *in* respectively *out* property, which are also new properties.

Some properties that the old version of EESim used have been removed. The property defining what frequency a simulation should run in have been removed for two reasons. Firstly, since *SyncSim* does not have any concept of frequency of a simulation and only cares about the change

of processor cycles, not how much time a cycle would occupy, there is no need to keep the property specifying a frequency. Secondly, EESim no longer uses a test-bench written in VHDL, that specifies a particular frequency.

The old version also had properties related to compilation of code to run on the MIPS model, but as EESim, now, is independent of the MIPS model these properties have been removed.

2.3 Evaluation

This section will discuss the value of the solutions to different problems in EESim as well as other design choices that were made.

2.3.1 Code layout

The changes made to the original code has resulted in a cleaner and more easily understood code, due to several factors. Unused blocks of code has been removed and the remaining code is more well commented. The files containing the actual EESim source code has been separated from all files having to do with the designs, which makes it easier to understand what is a part of the simulator and what is a part of a design for the simulator. The previous version had EESim's source code mixed with the MIPS model.

2.3.2 Rewinding

A feature that EESim lacks compared to SoftSim is the ability to rewind a simulation for one or more clock cycles. This feature was intentionally left out of EESim for a reason. The deciding factor in not including the rewind function is that the possible solutions would limit what type of designs could be used in NCSim, or demand too much in terms of processor power.

Rewind in SoftSim saves the history of all parts so that it is possible to go back to a certain point in time and retrieve the values for that particular cycle. When run or step is invoked after a rewind SoftSim will resume simulation from that point, that is, one cycle can be simulated more than once.

The problem with using this method with EESim is that it is possible for a part of the design that is simulated by NCSim to be in different states. Whether such a state exists or not is unknown to EESim. The state can depend on some previous cycles so just rewinding the requested number of cycles will not set the right state. To make this method work, EESim would have to have more knowledge about the contents of components simulated in NCSim.

To solve this problem all designs could be forced to show all registers implemented in VHDL to EESim, or even demand that all registers should be implemented as Java classes. The downside to this is that it would be much more work to use existing VHDL code with EESim when all registers need special treatment. This means that standard VHDL code would not work with EESim without modifying the code.

Another way — and the best in our opinion — to implement this is to store all out-signals of all synchronous parts for each rewinded cycle. If we run the simulation to $t = t_0$, rewind some cycles and then start to step forward again no new simulation will be done but the stored values will be used as long as $t \leq t_0$. All states of components simulated in NCSim will have the correct values for continuing the simulation after t_0 , since that is where NCSim stopped its simulation. Because the simulation will only be simulated forward in time there will be no problem with internal states of components.

Another solution, which would work in all cases and be easy to implement, is to reset the simulator and re-simulate up to the point to which the rewind placed the simulation. But this solution is, of course, practically impossible, since restarting everything takes a lot of time. This method was used by the old version of EESim.

2.3.3 NCSim synchronization

As stated in section 2.2.4, EESim generates a rising edge on the clock signal to make synchronous VHDL components simulated in NCSim react. This is a quite common way to model synchronous components, but sometimes one may wish to have components that react to a falling edge or possibly both. However, EESim does not support this and while it should be possible, no consideration have been placed on this issue.

It is possible that some problems or undefined behaviour may arise in some specific VHDL design because of this. But all tests that have been run have functioned as expected.

2.3.4 Design XML-file changes

The goal, as stated previously in section 2.2.5, of the new XML-file properties was to enable the user to add VHDL components to a design without writing any Java code for that particular component. This goal was reached without making the XML-file much harder to write. Two of the properties, however, are a bit more specific than was desired. These are the properties *hdlvar* and *cdslib* that specifies the paths to design files (*hdl.var* and *cds.lib*) for the VHDL code (see the EESim manual, Appendix A, for details). The properties are passed on to NCSim which requires them to be absolute paths, while the intention was to have them as relative to the location of the design XML-file. If SyncSim made it possible to get the absolute path to the XML-file, it would have been possible to state the paths relatively in the XML-file. Unfortunately, there is currently no way to get this path from SyncSim. The absolute paths will need to be changed if the design is moved around in the filesystem, which is not the optimal behavior. Values of other properties are more concise and are not affected by the location of the design files.

2.3.5 EESim dependencies

Using NCSim as a simulator for the VHDL parts of a design incurs some dependencies on which platform EESim can be run. Naturally NCSim itself will need to be supported on the platform where EESim is going to be used. This is not as bad as it may seem as NCSim is supported on a number of different platforms [6], but it still limits EESim compatibility compared to SoftSim. The other factor which constrains EESim's portability is the usage of POSIX [11] message queues, which requires a POSIX compatible platform to function. It should also be noted that EESim has only been tested on Solaris [12], so no guarantee can be given that it will function even if the two above mentioned requirements are fulfilled, although no possible problems have been anticipated.

2.3.6 Test case - running a mixed VHDL/Java MIPS design

During the development of EESim, the simulator was tested with a couple of small and simple designs just to make sure that different features worked as they should. These designs were written with both mixed VHDL and Java code as well as only VHDL code. By using these trivial designs it was easy to debug and test different parts of EESim. However, the goal of EESim was to be able to run mixed VHDL and Java models of arbitrary size and complexity so some sort of testing with a larger design was necessary to confirm that the simulator actually fulfilled this goal.

There exists a MIPS design written in Java for SoftSim. So for the final test of EESim the choice was made to replace some parts of the MIPS design with functionally identical parts written in VHDL. If EESim could simulate that modified model it would be considered as a strong indication that EESim actually was capable of simulating models of arbitrary complexity. In the final test, the register file and ALU was replaced with the equivalent VHDL components (figure 2.6 shows this). The model was then tested by running a few different programs in the MIPS design. These executed identically to how they functioned on the pure Java MIPS design. The mixed Java and VHDL MIPS design is available in the SyncSim CVS [7].

Chapter 3

PCC

3.1 Background

3.1.1 The C language

The C programming language was originally developed for the UNIX operating system on the DEC PDP-11 by Dennis Rithchie sometime around 1972 [3]. C quickly gained popularity and became the standard programming language for UNIX operating systems. It is still used as a programming language for situations that require low level programming.

The following example shows a program that calculates the first 20 fibonacci numbers written in C:

```
1.  int theArray[20];
2.  int last1;
3.  int last2;
4.  int index;

5.  int main() {
6.      theArray[0] = 0;
7.      theArray[1] = 1;
8.      index = 2;

9.      do {
10.         last1 = theArray[index - 2];
11.         last2 = theArray[index - 1];
12.         theArray[index] = last1 + last2;
13.         index++;
14.     } while (index < 20);

15.     for (;;) ;
16. }
```

This example will be referred to later in the text as the fibonacci program.

3.1.2 History of PCC

The first version of the Portable C Compiler (PCC) was written in the late 1970s by Steven Johnson [4]. This was one of the first attempts to create a C-compiler that was designed to be portable. That is, it should be easy to transfer the compiler from one computer architecture to another. PCC was successful in this as it was used on several UNIX platforms and also served as a reference compiler for other implementations until the C-standard was drafted.

In early 2001, PCC together with the UNIX Operating System, was released under the BSD license [14]. This meant that anyone could look at, and modify, the source code. Because of this, Anders Magnusson at Luleå University of Technology, chose this compiler for a project of his. But since PCC originally was written in the 1970s it lacked several modern compiler constructs. It did not support the latest ANSI C standard [2] and neither did it support code optimization. Anders Magnusson has implemented support for ANSI C in the compiler and has also added optimization functionality to it. PCC is still being developed by him. The changes from the original compiler to the new version are not documented so all background references will be made to [4].

3.1.3 PCC concepts

Since PCC is designed to be a portable compiler [4], the source code is modularized so that only a few files of the complete source code has to be modified to support a new computer architecture. The complete source code contains 42 source files (header and c-files), and of these only 6 are modified when porting the compiler. These files are referred to as the machine dependent files. Appendix C shows the file and directory structure of PCC for the important parts.

PCC is a relatively small compiler. The number of lines of source codes in the compiler core (without any machine dependent code) adds up to around 24000 in total. The MIPS port (see section 3.2) contains approximately 3000 lines of code, which is about the same as other PCC ports. This clearly shows that only a small part of the compiler needs to be considered when porting it to different platforms.

PCC is a multi-pass compiler, more specifically a two-pass compiler. This means that the compiler separates the tasks it performs into separate phases (passes). The C source code is read in and stored in an intermediary format in the first pass. This intermediary format is then used, in the second pass, to generate PCC's output. The output from PCC is assembly code for the target architecture it has been compiled for. Assembly code generated by PCC for the fibonacci program can be seen in appendix D. Figure 3.1 shows the layout of PCC.

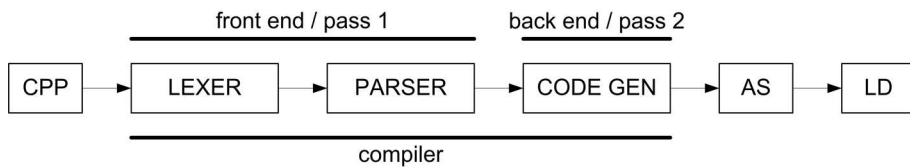


Figure 3.1: PCC layout

The first part, in figure 3.1, which is not included in the area marked “compiler”, is the C Pre-Processor. This part handles the Pre-Processor directives in the source code, something that is beyond the scope of this report. The second part, the lexer, reads in the source file and transforms the C code into a stream of elements, called tokens, that the parser can then translate to an intermediary format (see below). These two parts, the lexer and the parser, are said to be part of the compilers front end. The front end in a compiler is the part of the compiler that is dependent on the source language, that is, C in PCC's case. The next part is the code generation step. This part translates the intermediary format representation of the C source code to assembly code. In a compiler, this process is handled by a compiler's back end. The back end of a compiler is architecture dependent since it must generate correct assembly code for a specific computer architecture. The last two parts, as and ld, are the assembler and linker (also called loader). These parts are responsible for the translation of the assembly code to machine code and linking the resulting binary file into an executable file.

Intermediary format

PCC's intermediary format (the internal data structure representing the source code) is a binary tree, called a parse tree. Figure 3.2 shows an example of such a tree containing an assignment

from a char to an int. The nodes in the tree can be leaf nodes, with no child nodes; unary nodes, with one child node or a binary node, with two child nodes.

Nodes in the tree can represent both operations like addition, subtraction, shift, etcetera or storage types like registers, numerical constants and names.

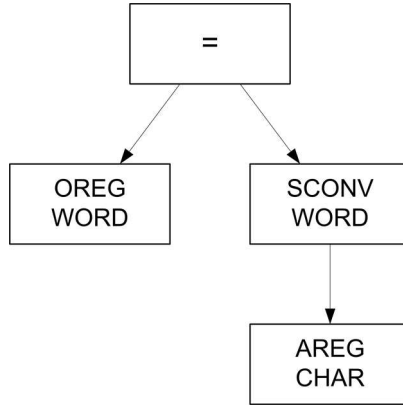


Figure 3.2: Example tree of an assignment

Translation table

A large part of any PCC port will be the translation table, which contains rules for how PCC should translate different nodes to assembly code. These rules look something like the following example from the MIPS port (section 3.2):

```

{ PLUS, INAREG|FOREFF,
  SAREG,  TWORD|TSHORT|TUSHORT|TCHAR|TCHAR,
  SCON,   TUSHORT|TSHORT|TCHAR|TCHAR,
          NAREG|NASR|NASL,      RESC1,
          "      addiu A1, AL, AR\n", }

```

Each rule is an entry in an array of all the translation rules for a particular PCC port. This array will be searched for matches to different node constructs when PCC evaluates the parse tree. The entries are instances of the *struct optab* structure and the block in the example above (and other similar ones in the translation table) is actually part of an assignment to one element in the array. So the comma separated list of constants and one string is the values for different members of the *struct optab*. These member values will be referred to as fields of *struct optab*. Fields that can contain more than one option consists of bit-fields where the values are or'ed together, like in the second field of the example.

The example above will be used to describe how these rules work.

The first field defines what operation this rule should match (addition in the example, that is, the + operator). In the second field two flags for the register allocator are given. The flags given in the example are very common. *INAREG* tells the register allocator that the result of the operation (addition in this case) should be saved in a register. The second flag, *FOREFF* means that this rule also can be matched if it is not necessary to save the result. There are several more flags that can be specified, but the inner workings of the register allocator is beyond the scope of this document and the interested reader is directed to [4]. The second line contains information about how the left sub-tree should look. First the shape (see below) is stated — a register in this case — and then a list of possible types of the left sub-tree is given. These types corresponds to the primitive data types of the C language. The third line gives the same information but for the right sub-tree. The fourth line contain two additional fields for the register allocator. The first field specifies how

many registers (if any) needs allocating and the `NASR` and `NASL` flags allows the register allocator to share the registers of the right respectively left sub-tree with the allocated register. That is, instead of allocating a new register as the `NAREG` flag specifies, the register allocator can — if possible — use either the register in the left sub-tree or the register in the right sub-tree in place of the register that would ordinarily be allocated when the `NAREG` flag occurs. This is something one would, of course, like to do as often as possible since it reduces register usage. The last line contains a string with the assembly code that should be generated for this particular node. Letters in the string that are capitalized are worth noting, since these are macros that, when expanded, will generate the correct register names, numbers or values. In our example, the `A1`, will result in the register name of the (possibly) allocated register. The `AL` will give the register name of the left sub-tree, and `AR` expands into the value of the right sub-tree.

Shapes

All leaf nodes will have a shape that determines what kind of value they store. They are used in the translation table, as described above. Table 3.1 lists the shapes as well as a description of each one. `SAREG` and `SBREG` are available so that the compiler can recognize two different types of registers, for example, general purpose registers and floating point registers. The `SOREG` shape contains a register with an address in it and an integer value that is an offset to this address. `SOREG`'s typically appear when variables are stored on the stack. `SANY` is used as a wild card in translation rules where the shape of a node is of no importance.

Table 3.1: Node shapes

<code>SAREG</code>	A register of type A.
<code>SBREG</code>	A register of type B.
<code>SOREG</code>	A register and an offset.
<code>SNAME</code>	A global name.
<code>SCON</code>	A numerical constant.
<code>SANY</code>	Any shape.

3.1.4 The MIPS architecture

When porting a compiler to a new computer architecture a good understanding of the target architecture is necessary. We will here give an introduction to the MIPS architecture and pinpoint what is important when porting the compiler.

History

The ideas about the MIPS architecture was founded by a team at Stanford University lead by Dr. John Hennessy in 1981 [10]. A few years later the research resulted in the start of a new company, MIPS Computer Systems, Inc. Today MIPS processors are used in a large number of products; routers, digital cameras, printers, etcetera, but not that many desktop computers.

Overview

The MIPS computer architecture is a Reduced Instruction Set Computer (RISC) architecture. This mean that it has a focus to increase performance by reducing the complexity of the instructions. In contrast to Complex Instruction Set Computer (CISC) architectures that with its more complex instructions can handle larger tasks. With the MIPS RISC architecture the compiler can break down the high level language code in small task that corresponds to the instructions of the architecture, whereas a compiler for a CISC architecture wants to find use for the more complex instructions.

The MIPS architecture uses an instruction pipeline to get parallel execution of instructions for increased speed. In some special cases there is a delay, a delay slot, of the instruction execution in the pipeline. To not waste the processors resources an instruction that is independent of the delay is preferred to run in that slot instead of just leave empty (run a nop instruction), more about this below.

The word size of the MIPS architecture is 4 bytes, each 8 bits long.

Instruction set architecture

The Instruction Set Architecture (ISA) is the interface between the software and the hardware, and must be followed by all code to be run on the architecture. The ISA of an architecture includes definitions of all the instructions. To translate high level language code into MIPS assembly code only a subset of all instructions is needed. Firstly, the compiler can get the same result by using different combinations of instructions. Some instructions might never be used by the compiler because it uses some other instructions that can perform the same task. For an architecture it is better to have some instructions that are not used than missing an important instruction that results in less performance. Secondly, some instructions are not allowed nor desired to be generated by the compiler. Kernel mode instructions are not generated by the compiler, e.g. instructions that control the cache.

Data types

To help utilize different data types on the MIPS architecture there are some instructions that work on data with different bit widths. The ALU always work with a whole word even if only a half-word or a byte of the word is of interest. All registers in the register file is one word wide and the whole word is always read or written at once. It is when data is loaded from and stored to a memory address that different instructions can be used depending on the size of the data. Word, half-word and a byte all have different instructions both for load and store. For example, to load the second byte from a word in memory and put it aligned and ready to use in a register only one instruction is needed. This is a big performance increase for loads and stores compared to if extra instructions were needed to prepare the data.

The architecture also supports loads of half-words and bytes with sign extend for signed data.

Delay slots

Delay slots arise in two different cases on the MIPS architecture, for loads and for branches/jumps. A data load is performed in the pipeline stage after the ALU, hence data loaded by one instruction is not available by the following instruction in the same clock cycle. The data is first available to the second instruction, with help from forwarding, after a load. For branches and jumps the new address is calculated first in the ALU pipeline stage, hence the following instruction will be fetched from the address of the old program counter. If the instruction was a jump or if the branch were true the second following instruction will be fetched from the address of the new program counter.

A no operation (nop) instruction can always fill these delay slots. The code will be correct but it is not an optimal solution. The best way to handle these delay slots is to reorder the instructions so that an instruction that is independent of the offending instruction is moved to the delay slot and only in extreme cases, when no independent instruction is found, use an nop instruction.

It can also be desired to have the option to turn off instruction reordering, e.g. code with instructions that has been reordered due to delay slots will have an unknown result when run in the none pipelined MIPS model for SyncSim.

3.1.5 Endianness

When data is stored in the processor (caches and registers) and the memory the bytes within a word can be in different order, different endianness. The most significant byte can be stored on the highest or the lowest memory address of a word, these two approaches are called little- and

big-endian respectively (see figure 3.3). The rest of the bytes will follow and end with the least significant byte at the lowest memory address for little endian and at the highest memory address for big endian. If we look at the bit level each bit within a word stored with little-endian can be read consecutive while with big-endian bits can only be read consecutive within each byte of the word.

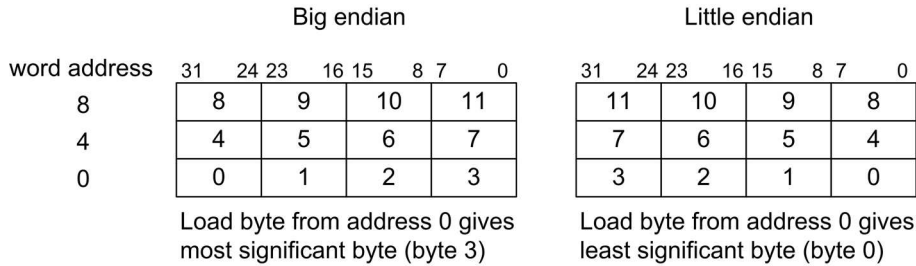


Figure 3.3: Little- and big-endian

3.2 Implementation

In this section the process of porting PCC to the MIPS architecture will be described. An already existing port for the x86 architecture [15] was used as a template of how a PCC port can look. The effort was made to reuse as much code as possible from the x86 port, to make the porting easier and less time consuming.

This section also, briefly, describes the implementation of a MIPS assembler.

3.2.1 Endianness in the MIPS port

Different versions of the MIPS architecture support either big-, little-endian or both. The MIPS model for SoftSim is a big-endian architecture so the MIPS port of PCC supports big-endian as default. This is, however, easily changed since it is only a flag in a header file which determines which endianness the compiler will have. So making the MIPS port a little-endian compiler would require no more than to change the value of the flag and to recompile PCC.

The point at which endianness becomes important is when C operations that generates load or store instructions of less than a word occurs. The following example shows such a scenario:

```
int a; /* 32 bit data type (a word) */
char b; /* 8 bit data type (a byte) */

a = 42;
b = a; /* Implicit type cast of int to char */
```

The second assignment (`b = a;`) results in a load from a memory position containing a word with the value 42. The generated load instruction will fetch one byte from the memory address of the word. If we don't generate the load instruction according to what endianness the word was stored with we might end up with the wrong value (0) in `b` instead of the value 42 which is what we want.

3.2.2 Translation table

Conversions

Conversions arise when the C code contains type casts, either explicitly or implicitly stated. Explicit type casts look like: `int a = (char)b;`. Implicit type casts happens when an assignment

between to different primitive data types occurs, as in the example above.

Conversions only need to be considered when storing to and loading from memory. The only way to handle this kind of conversions with MIPS assembly instructions is by using load and store instructions of different kinds. The MIPS instruction set contains instructions for both loading and storing bytes, half-words and words. These data types can be either signed or unsigned, and different instructions exist to distinguish that. There is, for example, an instruction for loading an unsigned byte and another instruction for loading a signed byte. Similar instructions exist for half-words and words.

Sizes of the different data types can be found in table 3.2. The unsigned versions of all data types have the same size as their signed counterpart.

Table 3.2: Data types, their sizes and names in the translation table

long long	64 bits	TLL
long	32 bits	TWORD
int	32 bits	TWORD
short	16 bits	TSHORT
char	8 bits	TCHAR

We will consider the following code excerpt as an example of how the conversion entries in the table look:

```
{ SCONV,      INTAREG,
    SOREG,    TWORD,
    SAREG,    TSHORT,
    NAREG,    RESC1,
    "        lh A1, ZA\n    nop\n", }
```

The example is a conversion from a long to a short. The first field of the entry, `SCONV`, is the conversion operation. The next field of interest is the third field, which contains the `SOREG` flag. Oregs (offset registers) typically represents variables that are stored on the stack. So, the third and fourth fields indicate that we have an `int` variable that is being type casted. In this case the third line specifies that the conversion result should be a short stored in a register. The fourth line is not of any interest here. The fifth, however, shows the assembly code necessary to convert an `int` to a short. This is, as stated previously, a simple load instruction. `lh` stands for load half-word. The macros in this case is simply first a register name and then a register with an offset. This last line also demonstrates how load delays slots are handled in the PCC port (that is, by inserting a `nop` (No Operation) instruction after the load instruction).

For the sake of completeness the following example displays how the same conversion would look when the result is an unsigned short.

```
{ SCONV,      INTAREG,
    SOREG,    TWORD,
    SAREG,    TUSHORT,
    NAREG,    RESC1,
    "        lhu A1, ZA\n    nop\n", }
```

The only difference is the data type of the right sub-tree and the instruction used in the assembly code line. The `u` in the instruction stands for unsigned.

The long long data type needs some special consideration because it occupies more than one register. One example of this is when we convert a long to a long long.

```
{ SCONV,      INTAREG,
    SOREG,    TUWORD,
    SAREG,    TLL,
```

```

NAREG, RESC1,
"      lw U1, ZA\n"
"      move A1, $zero\n", }

```

What is special here is that we use a U1 macro to get the register used to store one half of the 64 bits of the long long value (lw stands for load word) . The register containing the other half of the long long value is accessed with the A1 macro. The second line of the assembly code is also a bit unorthodox. The \$zero register always contains the value zero so it is used to zero out the unused half of the long long. Since a long is only half of a long long we will never have anything to store in the most significant half of the long long in this type of conversion, which is why we put the value zero in the unused half.

The translation table contains some more special cases as well as a lot more entries for conversion operations, but these will not be covered here, since they are similar to the examples given. The interested reader is referred to the source code for more details.

Assignments

The translation table entries for assignments contain all rules necessary for handling the assignment operator, '='. All assignment rules are assignments from a register to either another register, an oreg or a name. Other types of assignments, like an assignment from an oreg to another oreg will be converted to an assignment of the above mentioned form. This is done with the help of rules written for the purpose of converting leaves to registers. These rules have a node type of OPLTYPE. Such an entry looks like:

```

{ OPLTYPE,      INTAREG,
  SANY,        TANY,
  SOREG,       TCHAR,
  NAREG,       RESC1,
  "           1b A1,AR\n      nop\n", }

```

This entry states how an oreg containing a char shall be converted to a register. Several more entries exist in the translation table which covers all possible combinations of data types and shapes.

So, by these conversions the number of assignment rules are significantly decreased. Although there is still a lot of them. The following example entry specifies how a register is assigned to a name (global variable):

```

{ ASSIGN,      INTAREG,
  SNAME,       TWORD|TPOINT,
  SAREG,       TWORD|TPOINT,
  NAREG,       RRIGHT,
  "           1a A1, AL\n      sw AR, 0(A1)\n", }

```

These type of assignments will, for example, be generated for all the assignments in the fibonacci program, where all the variables are global.

Considering that we have three data types (char, short and int) we need one entry for each (the above covering the case of an int). We moreover need rules for names and register to register assignment as well as some special rules for long long assignments. The long long data type is handled in a fashion similar to ordinary data types, except that more instructions are needed because of the two registers used. For the sake of clarity one such rule is included below:

```

{ ASSIGN,      INTAREG,
  SOREG,       TLL,
  SAREG,       TLL,
  0,          RRIGHT,
  "           sw UR, UL\n"
  "           sw AR, AL\n", }

```

Branches

The branch instructions on the MIPS are combined comparison and branch instructions, which makes it necessary to have one rule for each type of comparison (for example, `<=`, `>=` or `==`). One rule, for the `==` comparison, looks like:

```
{ EQ,   FORCC,
      SAREG, TANY,
      SAREG, TANY,
      0,     RESCC,
      "      ZQ\n", }
```

The macro `ZQ` is what generates the actual branch instruction, depending on the nodes operand (`EQ`). The second field, containing the flag `FORCC`, simply specifies that it is a comparison instruction. Other branch rules are very similar. For example, the less than (`<`) branch instruction looks like this:

```
{ LT,   FORCC,
      SAREG, TANY,
      SAREG, TANY,
      NAREG|NASL, RESCC,
      "      sub A1, AL, AR\n      ZQ\n", }
```

This is the kind of situation that occurs in the while loop, at line 14, of the fibonacci program, for example. It is necessary to make a subtraction before the comparison, since the MIPS instructions only compares if a register is less than zero. The compiler always generates a nop instruction after the branch to fill the branch delay slot.

3.2.3 Calling convention

A calling convention is the way in which subroutines should be called in assembly code. That is, how local data, registers, etcetera should be saved on the stack. This is not something that is directly defined by the architecture. It is, as the name suggests, an adopted convention for calling subroutines. Both compilers and assembly programmers should follow the convention to make it possible to link handwritten code with compiler generated code.

The implemented calling convention is based on the description in [1] and has been compared with the output from GCC to insure that it is compatible with other compilers.

Figure 3.4 shows the layout of the stack for a subroutine.

Arguments to a subroutine are first (if possible) placed in special registers called `a0`, `a1`, `a2` and `a3`. If the number of arguments are larger than four or if any argument is larger than 32 bits, the argument is pushed on the stack. This is seen at the top of figure 3.4. Arguments are pushed by the calling subroutine, commonly referred to as the caller. After arguments have been stored (either in register or on the stack, or possibly both) control is given to the called subroutine, referred to as the callee. The callee then stores the return address and the value of the old frame pointer. After this, local variables and registers that needs to be saved are stored. Figure 3.4 shows where the frame pointer (FP) and stack pointer (SP) points in memory.

The PCC core (the machine independent part of PCC) does not really support subroutine arguments sent via registers. The MIPS PCC port therefore includes a workaround, which makes all callees push the contents of the argument registers to the stack before doing anything else. This is, however, not that serious since the calling convention [1] specifies that room should always be allocated for all four argument registers. It is interesting to note that GCC actually generates the same kind of code in this situation. The reason for this is unknown.

Return values from subroutines are stored in register `v0` and `v1`. If a return value is larger than what `v0` and `v1` can contain (64 bits) it should be stored to a position in the callers stack. This is not implemented in the PCC port (see section 3.3.1 for a discussion of this), which only support return values stored in register `v0`.

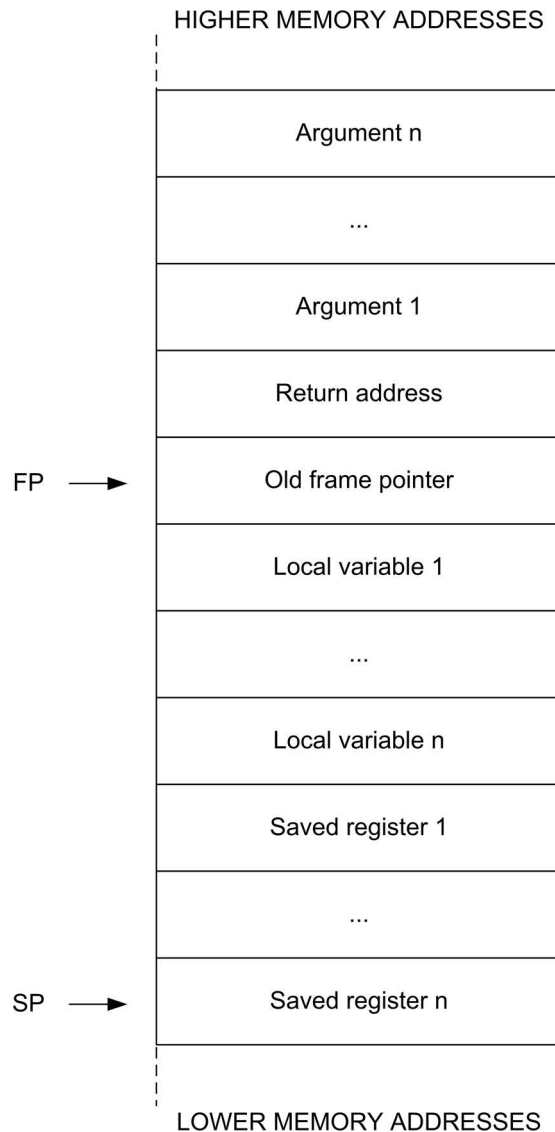


Figure 3.4: Stack layout in accordance with calling convention.

3.2.4 MIPS assembler

The next tool in the complete flow from source code to a compiled binary file after the compiler itself is the assembler. Since the compiler produces MIPS assembly code the assembler should be able to convert this to a binary format. The implemented assembler support ELF [16] as its output format. For the purposes of this report ELF can simply be considered as some sort of binary format. No more details than that will be considered. The assembler is called MAS; short for MIPS Assembler.

MAS uses Flex [17] and Bison [18] to create the lexer and parser of the assembler. The assembler is not complete at this time. It can compile assembly code, but lacks some functionality. The limitations of MAS and necessary improvements are discussed in section 3.3.3.

3.3 Evaluation and results

The final result of the PCC port is that it is able to generate code which can be used with the MIPS model for SoftSim. An assembler has also been partially implemented that can create ELF files from MIPS assembly code. Some important points of the PCC port and assembler are discussed in more detail below. Lastly, there is a section describing our experience with porting PCC.

3.3.1 Missing data type support

C structures (and unions) are not supported as return values from subroutines or arguments to subroutines. The choice to omit support for this was taken because of the relatively rare occurrences of structures sent to or from subroutines. Usually programmers send a pointer to a structure instead of the whole structure as an argument. This is done because the structure would have to be copied into a new memory position if it was sent as an argument, which can be ineffective if dealing with large or many structures. Adding support for structures in arguments or return values would, unfortunately, not be that easy. It includes a lot of fiddling with memory locations, which is quite complicated. The structure of the existing code would not have to be changed to incorporate the new functionality.

Floating point support is also lacking in the PCC port. This is mostly because the MIPS model in SyncSim does not support floating point itself, which means that there would have been no way of testing floating point support. Also, since the MIPS model is the primary target for the compiled code it was unnecessary to support something in the compiler which would not be used. It would require some work to add support for floating point, but as with structure support described above it is something which could be added to the existing code without any restructuring.

The final unsupported data type is long long. Previous parts of the report have described it as if it was working, which is not the complete picture. Mostly, usage of long long will fail except in some manufactured cases. This is because of the existing register allocator, which cannot allocate two registers (which is needed to store a long long) to a single variable. A new register allocator is in the final stages of development by Anders Magnusson and it will solve these problems. So the code for long long support in the PCC port will, with minor changes, work with the new register allocator.

3.3.2 Optimization

As it is now, PCC can optimize the code to work directly with registers instead of the stack for local variables. This can increase the efficiency a lot in many cases. What PCC doesn't support but which would be nice when generating code for the MIPS is the capability to reorder instructions. As described in section 3.1.4 load and branch/jump instructions cause delay slots. If the compiler could reorder instructions to fill these slots with useful instructions, instead of no operations as it does now, performance could be increased. To make PCC capable of this would require significant changes to the machine independent parts of the compiler, and this is the reason it was left out of the MIPS port. It would, however, be a useful feature to include in PCC in a generic way so that different types of instruction reordering could be easily implemented in machine dependent ways when porting PCC to different computer architectures.

3.3.3 The assembler

The features lacking in MAS is most importantly handling of relocation. That is, support for global names and labels. Relocation is the process of translating names and labels to offsets in the binary file, and without it the assembler is severely limited since labels and names cannot be used. The reason relocation is not supported is simply from lack of time. It is something which could be added to the existing code structure with relative ease. No problems has been foreseen for that.

Another thing that needs to be improved is the handling of incorrect instructions in the assembly code. This is not a problem with the code that the PCC port generates, since it will always be correct, but if MAS is to be used as a stand-alone assembler it must be able to detect and report errors in the assembly code. As with the problem described above, this should fit into the existing code structure without any major problems.

3.3.4 Porting

Since we haven't ported PCC to any other platforms it is not possible to estimate the degree of difficulty in porting PCC to the MIPS compared to other platforms. There are a few areas which proved challenging which will be discussed in this section. The most time during the work of porting PCC was spent on understanding PCC and reading the compiler's source code. Now that we have a good understanding of the compiler, it would be much less time consuming to make another port of PCC.

The MIPS port was originally based on a x86 port, as mentioned elsewhere, which saved us a lot of time, since we could work on one or a couple of areas at a time and still have the structure of the whole code ready. It was also helpful to see how some problems were addressed in the x86 port even though they differed from the MIPS's peculiarities.

It took a lot of time to write the entire translation table, but it was easy to add entries as we went along. PCC can report when some rule it is trying to match doesn't exist, which is how some rules were recognized as necessary.

Most effort was spent on implementing the calling convention, including function calls. Both the workaround for PCC's lack of support for argument passing via registers and the support for arguments itself caused difficulties. The number of actual lines of code isn't that big, but the process of figuring out how to write the code took time. This is true about the work as a whole. A lot more time was spent thinking about how to solve problems than to actually write down the code that solved them.

To sum up our experience of porting PCC to the MIPS one can say that with a good knowledge of the compiler (which we lacked) and a thorough understanding of the target architecture (which we had), it is not that much work to port PCC. Except if the target architecture needs some special functionality from the compiler that PCC doesn't support.

Chapter 4

Conclusions

The purpose of this thesis was to enable SyncSim to simulate designs with VHDL components and to create a compiler capable of generating code for use with the MIPS model for SoftSim. Both these goals were met.

EESim gives SyncSim the ability to co-simulate designs with Java and VHDL components. EESim could be used to complement SoftSim in courses that use SyncSim today to give students the opportunity to use a real hardware description language to model hardware. Converting a design from SoftSim to EESim is easy, since EESim can simulate Java components mixed with VHDL components. This means that a single Java part of a design can be replaced with its VHDL counterpart without changing the rest of the design. The VHDL code do not have to be modified in any way to work with a design in EESim. Standard VHDL code can be used.

The MIPS port of PCC can replace GCC for most purposes when compiling C code for the SoftSim MIPS model. PCC meets the demands of being a small and easily, as far as compilers goes, modifiable compiler. The groundwork for an assembler has been made and could be used as a starting point for a complete implementation.

Bibliography

- [1] Gerry Kane and Joe Heinrich. MIPS RISC Architecture. Prentice-Hall Inc., 1992.
- [2] Brian W. Kernighan and Dennis M. Ritchie. The C Programming Language, Swedish Edition. Computer Press Förlags AB, 1989
- [3] Ram Kumar and Rakesh Agrawal. Programming in ANSI C. West Publishing Company, 1992.
- [4] S. C. Johnson. A Tour Through the Portable C Compiler. Bell Laboratories, 1977.
- [5] Erich Gamma. Design Patterns. Addison-Wesley, cop. 1995.
- [6] Cadence NCSim. <http://www.cadence.com>. (Accessed: 2006-01-09)
- [7] SyncSim. <http://sourceforge.net/projects/syncsim>. (Accessed: 2006-01-09)
- [8] Daniel Edmark and Håkan Evertsson. EEsim-The Evertsson Edmark Simulator. <http://bart.sm.luth.se/~danrah-0/project/>, 2004. (Accessed: 2006-01-09)
- [9] GCC. <http://gcc.gnu.org>. (Accessed: 2006-01-09)
- [10] Mips Technologies, Inc. <http://mips.com>. (Accessed: 2006-01-09)
- [11] POSIX.1 FAQ. http://www.opengroup.org/austin/papers/posix_faq.html. (Accessed: 2006-01-09)
- [12] Sun Microsystems. <http://www.sun.com/>. (Accessed: 2006-01-09)
- [13] Java Native Interface Specification. <http://java.sun.com/j2se/1.4.2/docs/guide/jni/spec/jniTOC.html>. (Accessed: 2006-01-09)
- [14] Caldera License. <http://www.tuhs.org/Archive/Caldera-license.pdf>. (Accessed: 2006-01-09)
- [15] Intel. <http://www.intel.com>. (Accessed: 2006-01-09)
- [16] System V Application Binary Interface. <http://www.caldera.com/developers/gabi/latest/contents.html>. (Accessed: 2006-01-09)
- [17] Flex. <http://www.gnu.org/software/flex/>. (Accessed: 2006-01-09)
- [18] Bison. <http://www.gnu.org/software/bison/>. (Accessed: 2006-01-09)

Appendix A

EESim Manual

The readers of this document are supposed to be familiar with Cadence NCSim, SyncSim and the work procedures of creating an own design for the SoftSim simulator.

A.1 Introduction

EESim is a simulator that lets the user simulate a hardware model written in different languages. This simulator can not be run on its own; it is a simulator core that is run within the general simulator SyncSim that provides basic features and a GUI.

The hardware model to be simulated can be written partly in a HDL compatible with Cadence NCSim and partly in Java, or another language accessed through Java native interface. It is also possible to run a model only written in one language but in that case another simulator is probably a better choice. The part of the model written in a HDL is simulated by NCSim but the simulation is controlled by EESim, see figure A.1.

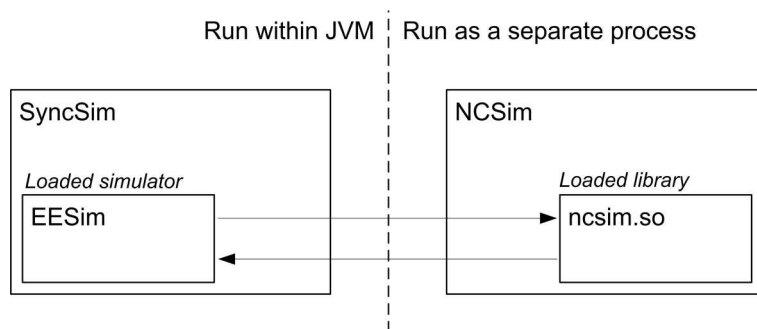


Figure A.1: Connection between SyncSim, EESim and NCSim.

A.2 About EESim

A.2.1 When to use EESim

When you want to simulate a hardware model written in different languages then EESim is the simulator to use. One possible scenario is if you have a design written for the SoftSim simulator and want to convert it to a HDL. Then you can change so that the design uses EESim and convert it part by part. This allows you test the design after each implemented part and not just a test of that part alone but the whole design.

A.2.2 Limitations with simulation in EESim

The same HDL code that runs in NCSim can be run with EESim but there are some limitations. You can not have any delays and only trigger on rising clock edges, i.e. signals can only change on a rising clock edge. The test bench might have to be rewritten or moved up to EESim.

EESim provides some basic test bench features. These are a clock signal and a reset signal that is set for one cycle right after the design is loaded. This means that you have to remove the clock- and reset signal from your test bench. You also have to remove or change all signals in the test bench that is not changed on a rising clock edge. Having a test bench running in NCSim not following these rules will give an undefined behavior. It is recommended to rewrite the test bench in Java.

A.3 Creating your own design

The parts of the design that are written in Java (if any) follows the same design rules as when creating parts for the SoftSim simulator. EESim's difference lies in how you connect VHDL components to the Java design.

To run any VHDL components in EESim you must have a compiled and elaborated VHDL design first. The VHDL model can consist of a single component or several interconnected parts. As long as it's possible to load your design into NCSim, it should also work with EESim.

If you have a design written for the SoftSim simulator you will have to inherit from EEPart and EESyncPart instead of from SoftPart and SoftSyncPart for EESim to work. If you are using parts from a SoftSim design, you should only need to change all references to SoftPart with EEPart and likewise with SoftSyncPart and EESyncPart for it to function in EESim.

A.3.1 The XML file

There are some new properties that need to be set in the XML file for the design, compared to the usual SoftSim designs. First you need to set the <simulator> tag to load EESim as the simulator. That looks like this: <simulator class="eesim.EESimulator">. Then there are two properties (cdslib and hdlvar) that are the paths to the cds.lib and hdl.var files for your VHDL design. It's probably easiest to give absolute paths to your files. EESim also needs to know the snapshot name of your elaborated design, which is specified with the snapshot property. The two remaining, new, properties are clock and reset. These should be set to the names of your clock respectively reset signals, or left as empty strings if your design lacks either signal. You need to specify the signal names with the whole architectural scope, e.g. :DUT:Clk. The following example shows how the simulator tag can look.

```
<simulator class="eesim.EESimulator">

  <!-- Complete path to file cds.lib -->
  <property cdslib="/home/user/.../vhdlsrc/cds.lib" />

  <!-- Complete path to file hdl.var -->
  <property hdlvar="/home/user/.../vhdlsrc/hdl.var" />

  <!-- Snapshot name for the elaborated VHDL design -->
  <property snapshot="worklib.top:rtl" />

  <!-- Name of the clock signal in the VHDL design -->
  <property clock=":Clk" />

  <!-- Name of the reset signal in the VHDL design -->
  <property reset=":Reset" />
```

... connections between parts ...

</simulator>

That covers the new simulator properties. All that remains now is to add your VHDL components and connect them together with the rest of the design. You should use the classes NcPart and NcSyncPart as the classes for your components. NcPart is used for asynchronous VHDL components and NcSyncPart for synchronous ones (see figure A.2). These classes are used so that no Java code needs to be written for parts which are defined in VHDL. However, there are four new properties which need to be set for these classes. The in and out properties are the same strings that you usually return from a parts in() and out() methods. To connect the in- and out signals of the Java classes to the actual VHDL signals you use the ncIn and ncOut properties. These should be set to a comma separated string with your VHDL signal names in the same order as you specified the signals in the in and out properties. So, for example, if you have one in signal to your VHDL component (lets call it :value) which you want to connect to the rest of your EESim design. Let's assume that you wish the name of the in port in the NcPart be called in1, and that it's 16 bits wide. You would then set your in property to "inJava:16" and the ncIn property to ":value". The properties for the out signals are set in the same manner. See the following example for an idea of what it looks like.

```
<component name="Adder" class="eesim.NcPart">  
  <gfx class="vhdl_counter.TextComponent" width="30" height="70" x="100" y="50">  
    <property text="+" />  
  </gfx>  
  <property in="in1:16" />  
  <property out="out:16" />  
  <property ncIn=":value" />  
  <property ncOut=":sum" />  
</component>
```

After that you make the connections between Java and VHDL components as you ordinarily would do it with a SoftSim design.

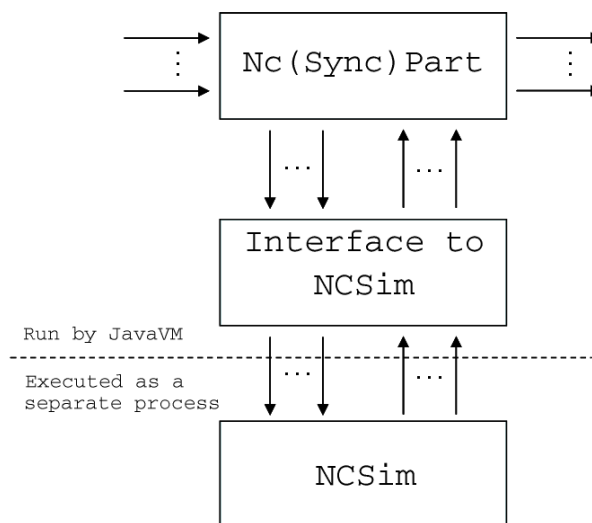
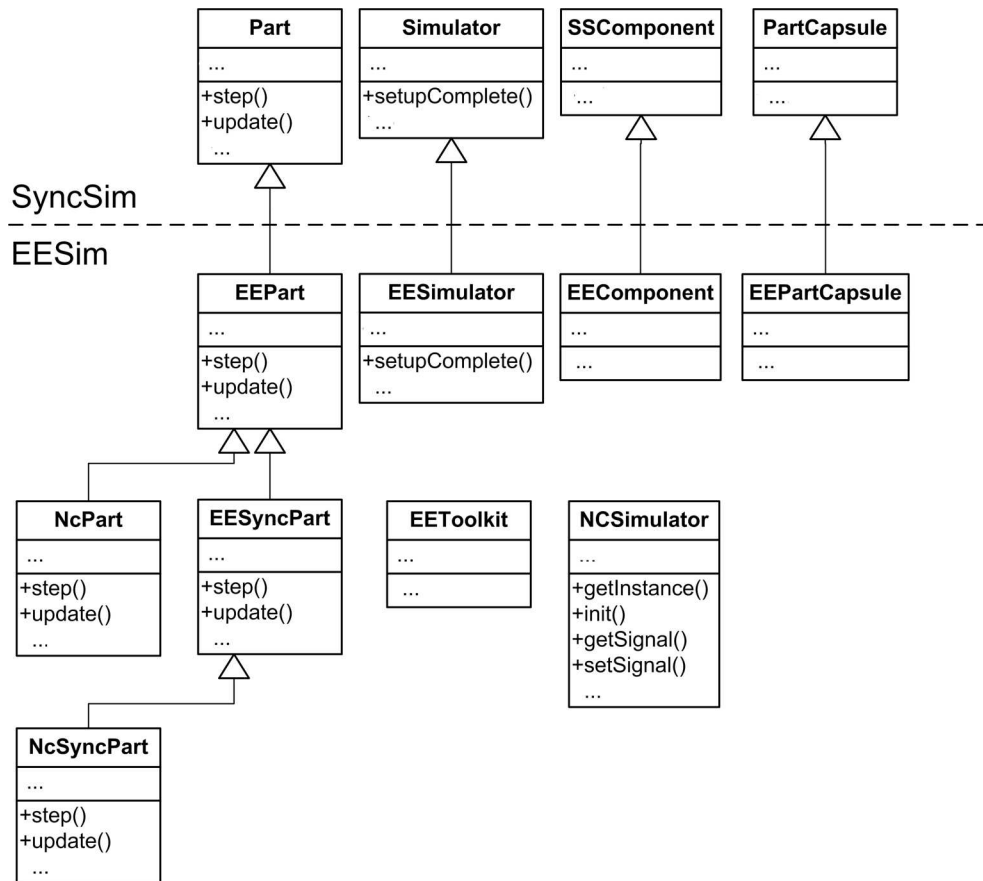


Figure A.2: Encapsulation of a VHDL part

Appendix B

EESim class diagram



Appendix C

PCC file structure

```
pcc/
  arch/                                     (location of machine dependent code)
    mips/                                   (mips dependent files)
      code.c
      local.c
      local2.c
      macdefs.h
      order.c
      table.c
    .../                                     (other PCC ports)
  cc/
    cc/                                     (source for pcc binary file)
      cc.c
    ccom/                                   (the compiler core)
      cgram.c
      external.c
      gcc_compat.c
      init.c
      inline.c
      main.c
      optim.c
      pftn.c
      scan.c
      stabs.c
      symtabs.c
      trees.c
    cpp/                                    (the pre-processor)
      cpp.c
      token.c
  mip/                                     (machine independent files)
    common.c
    match.c
    mkext.c
    optim2.c
    reader.c
    regs.c
```

Appendix D

Fibonacci assembly code

```
.data
.comm theArray,0120
.comm last1,04
.comm last2,04
.comm index,04
.text
.globl main
.align 4
main:
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $fp, 0($sp)
addi $fp, $sp, 0
.L15:
li $t0, 0
la $t1, theArray
sw $t0, 0($t1)
li $t0, 1
la $t1, theArray+4
sw $t0, 0($t1)
li $t0, 2
la $t1, index
sw $t0, 0($t1)
.L19:
li $t0, 2
la $t3, index
lw $t2, 0($t3)
nop
li $t1, 2
subu $t1, $t2, $t1
sllv $t1, $t1, $t0
la $t0, theArray
addu $t0, $t0, $t1
lw $t0, 0($t0)
nop
la $t1, last1
sw $t0, 0($t1)
li $t0, 2
la $t3, index
```

```

lw $t2, 0($t3)
nop
li $t1, 1
subu $t1, $t2, $t1
sllv $t1, $t1, $t0
la $t0, theArray
addu $t0, $t0, $t1
lw $t0, 0($t0)
nop
la $t1, last2
sw $t0, 0($t1)
la $t2, index
lw $t1, 0($t2)
nop
li $t0, 2
sllv $t1, $t1, $t0
la $t0, theArray
addu $t0, $t0, $t1
la $t2, last2
lw $t1, 0($t2)
nop
la $t3, last1
lw $t2, 0($t3)
nop
addu $t1, $t2, $t1
sw $t1, ($t0)
li $t2, 1
la $t2, index
lw $t1, 0($t2)
nop
li $t0, 1
addu $t0, $t1, $t0
la $t1, index
sw $t0, 0($t1)
subu $t0, $t0, $t2
.L18:
la $t2, index
lw $t1, 0($t2)
nop
li $t0, 20
sub $t1, $t1, $t0
bltz $t1, .L19
nop

.L17:
.L22:
.L20:
j .L22
nop
.L21:
.L16:
lw $ra, 4($sp)
lw $fp, 0($sp)
addi $sp, $sp, 8

```

```
jr $ra  
nop
```