1 Introduction

SyncSim is a general simulator for synchronous circuits that has been developed at Luleå University of Technology as a project in SMD153. It was mainly developed to replace SOLL, the MIPS emulator that for the moment is used in the computer organization and logic design course at the university. SyncSim is, unlike SOLL, a general simulator with a user friendly GUI and is not hard coded to emulate a specific CPU.

We have also implemented a subset of MIPS, which is fully capable of running simple c programs. This implementation is a good starting point for those interested in developing a complete MIPS model.

1.1 Purpose

The purpose with this report is to describe the development of SyncSim, explain its design and describe how to develop simulation models.

1.2 Scope

The report describes the development process of the simulator, explains its design, gives suggestions for improvements and presents the individual efforts and experiences of the group members.

2 Problem description and goals

In the course smd153 we worked with the project to create a simulator to replace SOLL as a teaching aid. Our goals were as described below.

2.1 Primary goals

The goal of this project is to create a simulator, which can be used to simulate various synchronous circuits. The GUI should be easy to use and responsive.

Every circuit that is to be used in a simulation will be composed of a number of components, which are coded in Java, and a description of how the components are connected to each other, which is described in a text file. The most commonly used components in a CPU will

be provided with the program. New ones, coded in Java, should be easy to add without any modification to SyncSim.

2.2 Primary features:

- Going one or several cycles forward and backwards.
- Pausing the simulation when a given condition is satisfied.
- Logging of data going in and out of components.
- Allowing the user to hide part of the circuit to get a better overview.
- Viewing interesting data of different components in the circuit. For example, one can view the memory contents of a memory component (in a number of ways, for instance as disassembled code, hex words, floats etc) and the register values of a register bank component.
- Combinatorial loop detection.

2.3 Secondary goal

Extend the GUI to make it possible to modify the circuit from inside the program without touching the text file.

3 Design

3.1 Overview

In order to maximize the reusability and extendibility of the program we have separated the GUI from the simulation as much as possible. The GUI uses the class Loader to load an XML file containing information about a certain simulation. Loader instantiates the particular Simulator subclass and other classes requested in the file and return a reference to the former to the GUI. The GUI then uses various methods declared in Simulator to control the simulation.

3.1.1 GUI

The graphical user interface in SyncSim is constructed in Borland JBuilder, which means that it contains a lot of generated code. The GUI code is in *SyncSimFrame.java* and it takes care of the menus, open/close/edit files with the built-in file editor and starting/stopping/creating the simulator.

The GUI also uses a configuration file, which is handled by *SyncSimConfig.java.* This file also contains code for a simple XML parser.

3.1.2 Breakpoints and Watches

The breakpoints and watches use the same parser for parsing expressions. The parser is generated with jacc and jflex, which are the Java versions of yacc and flex. The rules for jacc are written so that the parser returns an expression tree representing the expression.

The GUI part is made with a JTable using a customized TableModel. An AbstractTableModel is inherited and implemented so it will only allow the first column to be edited. When an expression is written in either the breakpoint list or the watch list the parser is called. The expression is only parsed one time, but breakpoints are evaluated every cycle and the watches every time the simulation pauses or stops.

3.1.3 Simulator

The class Simulator is an abstract class with methods for controlling the simulation. To add support for a new kind of simulation this class needs to be inherited.

Subclass should preferable use a new thread for doing the actual simulation work. Otherwise the GUI will not receive UI events and this means it will not be possible to pause a running simulation. It will also make the GUI less responsive, and graphics updates might be delayed.

Information is forwarded to the GUI in a way similar to how Listeners are used in Java AWT and Swing. The GUI provides a class that implements the interface StatusView. By using the method addStatusViewer the GUI adds itself to the list of event receivers. When a command is complete or other information needs to be forwarded to the user the corresponding method in StatusView is invoked on all StatusViewers added to the simulator. One should note though that the invocation might be made on another thread then the event dispatch thread, so any updates to the GUI must be handled carefully.

3.1.4 Loader

The Loader is responsible for parsing the XML file and creating the objects needed for simulation. The class contains an inner class LoadHandler, which handles the actual interpretation of the tags in the file.

LoadHandler contains two callback methods that are invoked by SAXParser at the beginning and end of each XML tag. The order of the invocations and the data supplied in the arguments is essentially the information contained in the file. A big part of this file is coping with errors in the input. Unfortunately the SAXParser is written somewhat badly, in the sense that it catches all Exceptions thrown from the callback methods. It tries to reconstruct the original information in the caught Exception, but if a cause was given it gets lost. This is the reason that some error messages might appear cryptic (to say the least) when there is an error in the XML file. This mechanism should probably be rewritten to give better feedback to the end user of SyncSim.

In the XML file, the tag `<simulator class=”xxx”>` must be present. The Simulator subclass with the name “xxx” will be instantiated and returned by `Loader.load()`. After the simulator tag, tags of type `component` and `gfx` may appear. The components correspond to objects of type Part in the simulation and the gfx tags correspond to visible objects of type SSComponent added to the main area of the GUI. Both tags imply that an

object will be created and the specific class name is supplied in the XML file. It is also possible to include properties in the tags. This is for instance usable when constructing one generic mux and using it with different bit widths on the ports and different number of inputs.

3.2 Our simulator

3.2.1 SoftSim

This subclass of Simulator is the coordinator of Java based simulations. It contains a thread waiting for commands and when a command comes from the GUI (in the form of a method invocation), the thread is woken up and starts the specified task. Every command has a corresponding inner class in SoftSim, StepAction is for instance the class that has a method for stepping one cycle in the simulation.

Basically, SoftSim is just a coordinator of the SoftPartCapsules. It stores them together with their name and gives out references to a specific one when needed. When a command is to be executed it is forwarded to all the SoftPartCapsules.

3.2.2 SoftPartCapsule

SoftPartCapsule is the layer that lies between the Part and the Simulator. When the simulator calls step or update in the SoftPartCapsule it checks if the input signals already are in the cache. If the cache is invalid, it fetches the signals from the next part in the call chain. SoftPartCapsule also handles the default tooltip, which is used whenever a part doesn't create its own tooltip.

3.3 MIPS model

The MIPS model we implemented is an attempt to make something useful for use in the program. It has not been extensively tested and one should not rely on it just yet (it has worked perfectly in our very small test programs though).

The model has been made to allow reuse of as much code as possible. Some parts are probably hard to find use for anywhere else, most notably the ControlPart which represents the control path of the MIPS. All the Parts are labelled with "Part" at the end of the class name, and all subclasses of SSComponents have the suffix Component. Some of the generic parts that could be usable in a large number of simulations are BusPart, ConstantPart, GeneratorPart, MergerPart, MusPart, SgnZeroExtendPart, ShiftPart and SplitterPart.

The MemoryComponent, CodeViewFrame, and DataViewFrame classes use the interface IMemory. This allows other simulations (apart from SoftSim) to use these GUI related classes. This would reduce the amount of work needed to do some other kind of simulation significantly if used properly. The same is true for the register file, where one could reuse RegFileComponent by letting the Part inherit IRegFile. Ideally all the visual components should be done this way, in most need to be updated are the Bus, Mux, and Alu.

The layout of our model is to a large extent copied from the simplest model in Soll. The Soll model seems unable to handle direct jumps (do not confuse this with branches!), at least it looks like that in the GUI, and this is probably the only difference in behaviour. Our modification is visible to the left in the GUI, the component with the text “merge” and the buses around it.

In order to simulate a program running in the model one uses the small program MipsInitializer. This program parses the `objdump` output to find a value for $GP and initializes the program counter to “main”. It does this by replacing $$GP in the XML file with the correct value for $GP and $$PC with the address of main. The stack is set to a default address 0x90000000 and a size of 4000 bytes. This needs to be changed if the program requires a bigger stack or needs the default address for something else.

4 Future Work

Although we are quite satisfied with SyncSim, there is always room for improvements and additions. Some of the features and additions we want mostly are:

- Connect breakpoints and the simulator
- Edit a design/model in the GUI.
- Better MIPS-model. Support for pipeline mode.
- Add functionality for the parts to write messages to the message window.
- Syntax highlighting in the source editor.
- Optimize the code, remove more bugs.

5 Individual efforts

5.1 Erik Byström

Has written SoftPartCapsule, the ALU and some of the minor Parts. Also the data path of the MIPS model, the breakpoint- and watches-window are written by him.

5.2 Lars Magnusson

Has made the Loader, Simulator and most of the Part and SSComponent subclasses (including the memory, register file and control path). Also did a large part of the debugging of the MIPS model.

5.3 Robert Selberg

Has made the graphical user interface, an xml parser for the configuration file, some Parts, project poster, a not included xml syntax highlighter for the editor and is the maintainer of the project site at syncsim.sourceforge.net.