Implementing a state snapshot for the EMCA Simulator

SAHAR SAGHAZADEH
Implementing a state snapshot for the EMCA Simulator

Sahar Saghazadeh

Department of Computer and Systems Sciences

Industrial Advisor: Tomas östlund
Examiner: Fredrik kilander

This master's thesis corresponds to 30 credits.
Abstract

The aim of this master thesis was designing and developing snapshot functionality for EMCA hardware simulator system which is used for implementing digital signal processing functions at Ericsson AB. The literature review contains the background knowledge on modeling, simulation, virtual machines and hardware simulators.

The simulator system has been reviewed and the design of the snapshot functionality for the system has been explained.

The performance of the snapshot functionality for saving and retrieving the visible state information has been evaluated. The results show that developed functionality works well in saving and retrieving the information.

Moreover, the effect of the invisible simulator states has been investigated. Some suggestions are presented for furtherworks.
Sammanfattning

Föreliggande rapport beskriver ett examensarbete med syfte att designa och utveckla snapshotfunktionalitet för EMCA hårdvaru simulator system som används inom Ericsson AB för implementering av digitala signalbehandlings funktioner. Kunskaps bakgrunden presenteras i form av en litteratur sammanfattning om modellering, simulerings, virtuella maskiner och hårdvarusimulatorer.

Simulatorsystemet har granskats och design av snapshotfunktionaliteten för systemet förklarats.

Prestandan av snapshot funktionaliteten har utvärderats i frågan om att spara och hämta tillbaka den synliga informationen. Resultatet av studien visar att den utvecklade funktionaliteten fungerar bra för att spara och hämta tillbaka information.

Vidare i denna studie har effekten av det osynliga simulator tillståndet undersöfts. Dessutom presenteras några förslag för fortsatta studier.
Acknowledgement

This thesis would have never been possible without help of several people.

I would like to express my gratitude to Per Gibson for trusting me and giving me the opportunity to work on this thesis at Ericsson AB. My special thanks go to my supervisor, Tomas Östlund for his valuable support and guidance. His knowledge and talent have always inspired me. I also want to thank Ruoyin Lai for his suggestions and follow up of the thesis steps.

I would like to express my sincere gratitude to my examiner, Fredrik Kilander for his valuable guidance and patience. Without his constructive feedback and insight suggestions this thesis would not have materialized.

Acknowledgements as well for all FlexTool team people at Ericsson AB who have been nice and helpful to me.

Last but not least, I would like to thank my family for always being there, supporting me and encouraging me how to deal with the problems with the confidence.

Sahar Saghazadeh
Table of Contents

Chapter 1. Introduction .............................................................................. 1
  1.1. Background .................................................................................. 1
  1.2. Problem statement ...................................................................... 2
  1.3. Purpose of the thesis .................................................................. 2
  1.4. Research Methodology ............................................................... 2
  1.5. Contribution ................................................................................ 3
  1.6. Structure of the thesis .................................................................. 4

Chapter 2. Extended background ............................................................... 5
  2.1. Modeling and Simulation ............................................................... 5
  2.2. Virtual Machines ......................................................................... 7
  2.3. Hardware Simulators .................................................................... 9

Chapter 3. State Saving Functionality ....................................................... 11
  3.1. Requirement of state saving .......................................................... 11
  3.2. State saving strategies ................................................................. 11
    3.2.1. CopyState Saving (CSS) ......................................................... 11
    3.2.2. Incremental State Saving (ISS) ............................................. 13
    3.2.3. Reverse Computation (RC) State Saving ............................... 14

Chapter 4. System Review ....................................................................... 15
  4.1. Flexible ASIC-FlexASIC .............................................................. 15
  4.2. Flexible ASIC Tool- FlexTools ...................................................... 16
  4.3. FlexASIC DSP ............................................................................. 17
    4.3.1. PCU .................................................................................... 17
    4.3.2. DAAU ................................................................................ 18
    4.3.3. CU ..................................................................................... 18
    4.3.4. Timer Block ........................................................................ 18
    4.3.5. PIF Blocks ........................................................................... 18
  4.4. Softsim Simulator ......................................................................... 19
    4.4.1. Simulator Core ...................................................................... 19
    4.4.2. Simulating Execution ........................................................... 19

Chapter 5. Snapshot Functionality of the Softsim Simulator .................... 21
  5.1. Inspiration and Implication for Design ......................................... 21
  5.2. Use Scenario ................................................................................ 21
  5.3. System Requirements ................................................................. 22
5.3.1. Functional requirements ................................................................. 22
5.3.2. Non functional requirements .......................................................... 23
5.3.3. An example of using the general commands of the debugger ...................... 23
5.4. Design Process ...................................................................................... 25
  5.4.1. Take the snapshot ........................................................................... 27
  5.4.2. Retrieve the Snapshot ..................................................................... 31
5.5. Example interactions ............................................................................ 33

Chapter 6. Evaluation ..................................................................................... 35
  6.1. Overall approach of testing ................................................................. 35
  6.2. Testing process of saving and retrieving snapshot of the state ................. 36
  6.3. Testing process of snapshot functionality to continue simulation ............. 40

Chapter 7. Conclusion and Future Work ....................................................... 45
  1.1. Conclusion ......................................................................................... 45
  7.2. Future Work ...................................................................................... 46

Chapter 8. References .................................................................................... 47

Appendix ........................................................................................................ 51
List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMCA</td>
<td>Ericsson Multi Core Architecture</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signalling Processor</td>
</tr>
<tr>
<td>PCU</td>
<td>Program Control Unit</td>
</tr>
<tr>
<td>CM</td>
<td>Common Memory</td>
</tr>
<tr>
<td>LPM</td>
<td>Local Program Memory</td>
</tr>
<tr>
<td>LDM</td>
<td>Local Data Memory</td>
</tr>
<tr>
<td>PIF</td>
<td>Peripheral Interface</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1: CSS Strategy: A copy of state variables is saved before each event processing [15].............. 12

Figure 2: ISS Strategy: A “change log” creates with each event which points to the old value of the state variables [15].............................................................................................................. 13

Figure 3: General FlexASIC System-On-Chip (SOC)........................................................................... 16

Figure 4: Block diagram of a general FlexASIC DSP ........................................................................ 17

Figure 5: Block diagram during execution EMCA DSP program.......................................................... 20

Figure 6: An example of using the commands of the debugger ............................................................ 24

Figure 7: A block diagram of the main transactions of snapshot functionality ..................................... 26

Figure 8: Saving and retrieving a simulation state by using snapshot and recover commands through the User Interface........................................................................................................................................... 27

Figure 9: The sequence of the state information in the snapshot file .................................................. 30

Figure 10: Block diagram of retrieving the state information from snapshot file to the system .......... 33

Figure 11: Block diagram of using the snapshot functionality for saving and retrieving the state information ........................................................................................................................................................................... 36

Figure 12: Example of testing saving and retrieving the simulation state after 40000 running cycles...... 38

Figure 13: Block diagram of using the retrieved snapshot for continuing debugging ....................... 40

Figure 14: Testing process of using the retrieved snapshot of the state for continuing debugging....... 41
List of Tables

Table 1: Sample tests of testing the snapshot functionality to save and retrieve the snapshot ............ 39

Table 2: The results of first experiment, saving and retrieving the snapshot of the state ....................... 52
Chapter 1. Introduction

1.1. Background

EMCA (Ericsson Multi Core Architecture) is a concept for implementing Digital Signal Processing (DSP) functions at EricssonAB in Kista, Stockholm. The concept is based on the integration of DSP cores on a single ASIC chip.

FlexASIC is a concept for integrating several DSP cores with shared memories on a single ASIC. It enables design of small, low cost and high performance solutions where reprogramming flexibility is a requirement. FlexASIC is supported by a complete set of software programming tools, Flexible ASIC Tools (or FlexTools for short) [24]

FlexTools are split in two software packages, SDK and DBG. SDK (Software Development Kit) contains all tools needed for building an application for a FlexASIC, while DBG package contains all tools needed for debugging the application like command line debugger, graphical debugger, API and a simulator called Softsim. Softsim is an architectural level simulator (cycle accurate) with debugging capacities that completely models all side-effects like pipe stalls, memory wait states etc. The simulator is the focus of this thesis work.
1.2. Problem statement

The current FlexTools simulator doesn't allow the designers and testers to execute backwards. Addressing this problem is the main focus of this discussion, by presenting a system prototype of snapshot functionality.

1.3. Purpose of the thesis

The aim of this work was to investigate, explore and prototype snapshot functionality for saving and loading the simulator state in order to:

- Make it possible to revert to an earlier state without executing the complete test case
- Step backward
- Suspend and resume a test case
- Share data between designer and tester

1.4. Research Methodology

Considering to the nature of the problem, this thesis work can be divided to three main steps: pre-studies, design and implementation of prototype, and system evaluation.

At the first step literature studies were done to get knowledge about similar works and related technologies to find possible solutions and make a prototype.
It was important to get full information about the structure and the components of the simulator system before starting the implementation. So in the next step the different components of the Softsim simulator system have been analyzed and the important parameters of simulation state were identified.

Based on the works done in previous steps, a prototype of the snapshot functionality was created for the system. So that by using snapshot functionality, a user can step backward in the simulation. Snapshot functionality saves the simulation state information to disk and retrieves it from disk back into the system.

The next step was system evaluation. I have conducted two evaluation experiments: The first experiment was designed in order to verify the ability of the snapshot functionality to save and retrieve the simulation information. The second experiment aimed to study using the retrieved snapshot for continuing the simulation debugging.

1.5. Contribution

The efforts were done as a part of my master thesis a project at Ericsson Company, Stockholm, Sweden. In this work, studying the concepts of simulation theories, virtual machines and hardware simulators were the contributions to computer science. Designing and developing the snapshot functionality for the Softsim simulator were the contributions to the software engineering field. I was mainly responsible for developing the snapshot functionality for the Softsim simulator. During the task I learned a lot about the design and architecture of the system.
1.6. Structure of the thesis

This report contains seven chapters:

Chapter 1 of this thesis contains the background of the research. The research problem and research goal are described in this chapter.

Chapter 2 provides a background of the topic, presents summary information of simulator platform and mentions the importance of the simulator systems.

Chapter 3 provides the background information about the state saving functionality.

Chapter 4 reviews the Softsim simulator system.

Chapter 5 focuses on actual implementation phase, describes system architecture, the technical aspects of the system, and how the snapshot functionality has been implemented.

Chapter 6 is dedicated to system evaluation and provides feedback about the performance of the implemented snapshot functionality.

Chapter 7 presents the conclusions of the discussion and some ideas for future improvements.
Chapter 2. Extended background

Since simulation is the process of designing a model of a real system, I felt a need to do a literature study first about the modeling and simulation theories followed by a study of virtual machines. Also since this study centered on how to make a snapshot functionality for a hardware simulator called Softsim, I finally made a discussion about the hardware simulators in this chapter.

2.1. Modeling and Simulation

Modeling is the procedure of creating a model which represents the structure and construction of some system of interest [1]. In the book "Discrete-Event System Simulation" (1996) [2], a system is defined as a group of interactive and interdependent components which is integrated together as a whole in order to achieve some purpose [2]. Modeling facilitates changes in the systems and it is a better way to quickly find out the changes in system behavior.

According to Bernard P. Zeigler’s idea [1] about modeling and simulation theories, a produced model should be as similar to the target system as possible and offer similar functionality and behavior while it should not be so complicated that it is impossible for the user to understand or maintain it [1]. In addition, the complex models need more resources to simulate, so we need to reduce the complexity of the models to be able to execute the system on our limited resources. Also, a well-developed model is a careful combination of realism and simplicity [1]. But Maria Anu in the article "Introduction to modeling and simulation" believes that the simplified model should still be valid and reliable. Verification and validation of the model should be done by executing the
simulator and the real system with known input and compare the output [4] [21]. This verification technique also let the analyst to find differences in precision between two modeling methods [21].

Since reconfiguration and experimenting with the actual system is usually more time-consuming, expensive and sometimes even impossible, studying the behavior of the real system can preferably be done by studying the model instead. In general, simulation is considered as a method to easier and faster assess the existing or proposed performance of a system over a long period of real time and with different configurations [4].

System simulation could be used before building a new system or changing an existing system to decrease unexpected bottlenecks, optimizing the use of resources and the system performance [3]. For instance, a telecommunication network simulation can be used to answer some questions such as: How will the network performance be changed if another routing algorithm is used for the network? What is the effect of a link failure on the network? [4].

Simulation should be chosen as a tool to analyze the system whenever there is a need to study the uncertainty in the system. However, simulation modeling of a real system could be complex and time consuming itself. Applying incorrect performance measures, existence of program bugs in the simulation, lack of a valid model, undocumented assumptions, unclear objective and lack of communications among the people who are engaged in the simulation study are a number of difficulties in the modeling process [4].
2.2. Virtual Machines

To a better understand the “Virtual Machine” concept, let’s think about the meaning of the “machine” from both system view and also process view [5].

A “Process Virtual Machine” is a platform that only supports a process and is expired when the process finished. However, a “System Virtual Machine” offers a complete continuing environment and supports an operating system with numerous processes at the same time [5] [6].

A virtual machine is used for developing the performance of a computer system. It can reproduce the hardware resources and provide several virtual hardware environments [6]. To implement a virtual machine, the developers add a software layer to the real system to avoid the hardware resource limits. In spite of the benefits of using virtual machines, they raise overhead on the host and cause lesser performance of both the host and guest compared to using physical machines [6].

The virtualized systems can be classified into three categories based on the common implementation issues:

1. Virtual operating systems which have been implemented to support multiple operating environments on a single computer system.

2. Virtual machines which are created by interpreted environments like Java which makes code effectively portable between the different instruction architectures.

3. Virtualized computer systems in the form of simulators to address the lack of real hardware.
While virtualization can be performed at different levels of abstraction, providing virtualization at the correct level to transparently support applications is crucial in practice. Operating system virtualization and hardware virtualization are two main methods to present application transparent virtualization. Here I am not going to talk about a specific product, to keep this review at a higher level of abstraction. [28]

Operating system (OS) virtualization technique virtualizes the operating system to split applications from the operating system so that the isolated applications can be executed in virtualized environments. For instance Parallels Workstation, Linux-Vserver and Solaris Containers are software products that provide the operating system-level virtualization. [28][29]

Hardware virtualization method virtualizes the underlying hardware architecture using a virtual machine monitor to isolate operating system from the hardware so that a full operating system environment and coupled applications can be executed in a virtualized environment.

Implementing any type of virtualization reduces physical space used by IT systems. Each virtualization technique provides its own benefits. For example OS level virtualization enables the organizations to do cost saving firstly because it requires fewer number of physical machines and lower power consumption. Virtualized servers are easy to be organized because if more virtual machines are needed, the same template can be used and loaded to the virtual environment. Although implementing virtualized operating system environment is useful for the organizations but if all the virtual machines go down this environment goes down along with it. So it is vital to have an efficient failover to make sure that if one virtual machine goes down, another virtual machine can be used instead. [30]

Since this thesis focuses on developing snapshot functionality for a hardware simulator, an overall description of hardware simulators is given in the following and the detailed description of the Softsim simulator will be addressed in chapter 4.
2.3. Hardware Simulators

Since running the real hardware repeatedly for evaluating purposes may cause harm to the device, a hardware simulator is a form of virtual machine which simulates the hardware circuits to evaluate the design of the hardware system. The simulator represents the actual system. It allows the developers to look faithfully into the hardware system without dealing with the real hardware. Since almost all system features can be integrated in a simulation model, simulating is suitable modeling approach when the system is large and complex.

Since a hardware simulator is a software-based tool, the developers are able to change the system easily. The developers can add some assumed features and functions to the simulator that might be impossible add to the actual system [8]. Also the cost and time needed for developing a simulator are less than for the real system. The hardware simulators are enhancing the quality of testing. The hardware simulator increases the scope of the testing, so that it enables the tester engineer to test at or beyond the variety of the certain engine parameters and verify the system at the failure conditions. [31]

Simulation of general hardware has three main stages:

1. Planning stage: In the planning stage the developer defines the problem, designs the related model, and plans a set of testing for making the simulation model.

2. Implementing stage: In the implementing stage, the developer initializes the variables so that the simulator begins from an identified state.

   The simulator executes the events and collects the generated data. These data are processed and interpreted to performance measures.

3. Testing stage: The experimental scenarios, planned in the first stage, will be verified in the testing part.
A hardware simulation can be considered as a flow of interaction between the entities. A simulation includes the following main components: [32]

**Entities:** Entities are the system objects that interact with each other. For example in a network simulation the entities are nodes, packets, simulation clock, etc.

**Resources:** The resources are the virtual assets shared through the entities.

**Activities:** The simulation entities employ in some activities that make the changes in the system state.

**Variables:** According to the scope of operation, the variables of the system can be categorized as global and local variables. The global variables are recognized by all the program entities. However the local variables are valid only in a limited range.

**Random number generator:** A simulator needs a random number generator to provide random numbers, because without using random numbers the results of the output would be the same for each simulation run.

The simulation performance is measured based on merits such as execution speed, cost, reliability, repeatability and scalability.

There are an extensive variety of simulation tools in the market. While each tool has its own strengths and weakness, the type of simulation development and the simulator user interface are two key elements that should be considered to choose the most suitable one.

The open source simulator tool might behave in different way by different users, while the developers are responsible for the closed source tools. And sometimes commercial software is better documented than open source software. [34]

The user interface of a simulator system can be a Command Line Interface (CLI) or a Graphic User Interface (GUI). CLI based simulators has most computing power while GUI based simulators allocate a part of their computing power to improve the user interface. So CLI based simulators are suitable to get results from a large body of simulation work.[32]
Chapter 3. State Saving Functionality

3.1. Requirement of state saving

The high cost of designing a hardware simulator leads to an increasing demand on using simulation systems with recovering ability. They allow changing the modules at run time. Saving and retrieving the simulation state is an important function of the hardware simulator. For example, a module can be interrupted and resumed later from the suspended state. Using a snapshot functionality helps communication between the developers [13].

3.2. State saving strategies

The operation to recover a previous state in a simulation is known as a rollback [27]. A usually used technique for employing rollback is “state saving” that stores the values of simulation state variables prior to an event conclusion and returns them by referring to these saved values [27]. Several techniques have been proposed to address the state saving issue. Three common methods will be discussed in the following sections.

4.2.1. Copy State Saving (CSS)

In the Copy State Saving (CSS) method an entire copy of the simulation state could be saved during simulator execution. Figure 1 shows a schematic diagram of CSS method. As it shown in figure 1, all values of the variables X and Y are saved in the snapshot. Saving full copy of state needs large amounts of disk space [15]. From a functional point of view CSS is a simple and precise technique. However, it slows down the simulator execution and is not an effective strategy for doing automatic state saving [14]. Since the number of rolled back events is usually less than the numbers of executions, it is often wasteful to store a full copy of each simulation state [14].
Figure 1: CSS Strategy: A copy of state variables is saved before each event processing [15]
3.2.2. Incremental State Saving (ISS)
The Incremental State Saving (ISS) technique addresses only the variables that have been modified or added to the simulation state. As figure 2 shows, in the ISS method, a change log that points to the old values of the state variables is created for each event [14]. To restore the old state, the state changes are undone by restoring the previous state values [14]. The ISS method is more efficient than the CSS method if not most variables of the simulation state have been changed [15]. It is a suitable method for doing automatic state saving.

Figure 2: ISS Strategy: A “change log” creates with each event which points to the old value of the state variables [15]
3.2.3. Reverse Computation (RC) State Saving
The Reverse Computation technique addresses re-computation of the earlier state so that a copy of the ended states has been saved after each event. This technique requires less disk space. However, it may need much more execution time than other techniques particularly when the event includes very small computations [17] [16] [15].
Chapter 4. System Review

4.1. Flexible ASIC - FlexASIC

The FlexibleASIC is an Ericsson’s concept of integrating several Digital Signal Processor (DSP) cores with shared memories on a single Application Specific Integrated Circuit (ASIC). The FlexASIC concept is based on the integration of all components on a single ASIC chip. Figure 3 shows the general FlexASIC System-On-Chip (SOC).

As figure 3 shows a flexible number of DSP cores can be used on a FlexASIC. Flexible memory size is supported by using Common Memory (CM) and due to the CM controller all blocks which are connected to it can access the memory simultaneously [24].

Depending on the application, a varying number of hardware Accelerators (ACC) can be used to handle intensive computational parts.

Also depending on the application, a number of ASIC Interfaces (ASIF) are available. They can be used for circulate information and Common Memory Interfaces (CMI) [24]. The Miscellaneous block (Misc) contains different interrupts such as timer interrupts, boot interrupt requests, resetting hardware and so on.
4.2. Flexible ASIC Tool- FlexTools

Flexible ASIC is supported by a complete set of programming tools which are called FlexASIC Tools. It has been used within Ericsson since 1996 in many projects and it is a truly embeddable, scalable, and vendor independent Multi-DSP Concept [24].

FlexTools are split into DBG and SDK packages. DBG contains all tools needed for debugging and SDK (Software Development Kit) contains all tools needed for building [18].

To improve the debugging facilities, FlexTools has been integrated with the Mentor Graphics EDGE debugger. This gives the user a unified software debugger environment based on Eclipse providing contemporary Assembler, C-debugger and Real Time Trace (RTT) tools and gives a complete build, simulation, debug and target debug environment [24]. The simulator is called “Softsim” which is the focus of this thesis.
4.3. FlexASIC DSP

According to the architecture of the FlexASIC, a number of DSP cores are integrated with shared memories that include the information systems which are required for the snapshot functionality. So in this part of the report I present more information about the common components of FlexASIC DSP shown in figure 4 to understand the data flow in the system.

![Block diagram of a general FlexASIC DSP](image)

**Figure 4: Block diagram of a general FlexASIC DSP**

4.3.1. PCU

The Program Control Unit (PCU) monitors the execution of the DSP. It reads the instructions from local program memory (LPM) and initializes itself and other units for execution. The PCU executes all program control instructions which include branches, calls, handling data transfer to the Common Memory (CM), handling the conditional execution and interrupt functions [3] [10].
4.3.2. DAAU
The DSP core has a Data Address Arithmetic Unit (DAAU) which generates address for Local Data Memory (LDM). The DAAU performed indirect memory accesses and address modifications by using a set of general address registers [3] [10].

4.3.3. CU
The Computational Units (CU) performs operations transmitted by the PCU. There are four CUs in a flexASIC DSP core. The different CU:s are capable of performing different parts of the instruction set.

4.3.4. Timer Block
The programmable timer can generate periodic interrupts based on multiples of the cycle clock [3] [10].

4.3.5. PIF Blocks
The DSP core can communicate with other DSP:s and units. Incoming messages are placed in a buffer in LDM [10]. Outgoing messages are sent through the Peripheral Interface (PIF) blocks which are used for connecting the peripheral units to the DSP core [3] [10].
4.4. Softsim Simulator

The Softsim simulator is a cycle accurate architectural-level simulator which completely models all side-effects like pipeline stalls, memory wait states, etc.

4.4.1. Simulator Core

The simulator core is the software component that does the actual simulation of the DSP hardware. A simulator core’s block corresponds to the hardware concept of a DSP block, such as the PCU, and contains resources, state information and instruction definitions. [3][18] A resource is an entity that can be inspected by the application, such as registers and memories. The blocks are implemented as a number of shared objects (.so-files) where each object corresponds to the hardware concept of a specific DSP block. [3] [18]

4.4.2. Simulating Execution

Softsim simulator calls basic_exec_cycle function to simulate a cycle of execution. A simulated cycle is split into several phases: [18]

- One startup phase for doing internal initialization within a block.
- One or several phases to perform the individual steps of the execution where each phase n may affect phase n+1. For example, in the PCU the phases represent the pipeline steps like fetch, decode and execute.
- One final phase to store the result of the simulated cycle.
These phases are implemented by calling functions `begin_cycle()`, `phase_function()` and `finish_cycle()` in order. [18]

![Block diagram during execution EMCA DSP program](image)
Chapter 5. Snapshot Functionality of the Softsim Simulator

5.1. Inspiration and Implication for Design

Snapshot functionality facilitates the communication between the designers and developers of the system. They can share the snapshot of the state to consult for solving problems.

The developer takes a snapshot of the simulation state and sends it to his/her fellow workers. Another developer retrieves the snapshot and gets access to the same simulation state.

A snapshot of the simulation state is a valuable source of the state information that can be retrieved any time and enable the developer to get access to the specific simulation state. If the simulation running is interrupted for any reason, the tester or developer can restart the debugger from the last simulation state.

5.2. Use Scenario

This section describes how the users are predicted to use the system. An example scenario in which snapshot functionality of the Softsim system could be used is as follows:
A scenario of Sarah Ravi

Sarah Ravi is young developer who works on the simulator system. She runs the simulator for a number of simulation cycles (e.g. 40000 cycles). While simulating she finds a problem and wants to consult with the developer, Petter, about the solution. She takes a snapshot of the simulation state and sends it to Petter.

Petter loads the snapshot file. He recovers the snapshot in to the debugger. He gets access to the same line of the source code to investigate the problem. By using the snapshot functionality, he avoids running the simulator for 400000 cycles again. Thus, the problem solving becomes easier and faster through using the snapshot functionality.

5.3. System Requirements

5.3.1. Functional requirements

The functional requirements present the functionalities of the system that have to be made successfully. They describe the user interfaces and interaction with the application that show how the designed snapshot functionality is used through the user interface.

The Softsim simulator has one main user interface. Softsim is a Linux based application and uses the Linux command-line terminal as the user interface.

Since the Softsim simulator already existed and the purpose of this thesis was to develop snapshot functionality for the simulator system, the previously designed commands and functions of the debugger were employed in this project. These commands are explained in section 5.2.3 by presenting an example.
5.3.2. Non functional requirements

The non functional requirements present the general properties and limitations of the system.

- The simulator system runs on the Linux operating system
- It uses a C developing environment on the Eclipse platform
- An assembler file 'pgm.asm' which was provided by Ericsson AB and contained randomly generated assembler program. This assembler file was assembled into an ‘.obj’ file through the Linux terminal by using the ‘flasm pgm.asm’ command. Then the pgm.obj file was loaded into the debugger by using the ‘loadobj pgm.obj’ command.

5.3.3. An example of using the general commands of the debugger

In this section I present a sample exercise to explain how to use the commands of the debugger through the user interface.

In this exercise I first start the debugger and run it for 20000 cycles. Then I show the memory mapping and line of the source code at the achieved simulation state. The steps of this sample exercise are shown in figure 6.
Starting the debugger by the 'fladb' command.

By running the debugger the registers and the memories values are initialized to zero.

Loading an object file in to the debugger by the 'loadobj' command.

In the previous step the debugger started but the debugger cannot be run yet, then I load the 'pgm.obj' to provide random value for running the simulator system.

-showsrc

As shown in figure 6 immediately after starting the debugger the source code line points to [0x000000] address. But the source code line will be changed later after running a simulation cycle.
Running a simulation cycle by the 'go' command.

\[-\text{go\ [nof-cycles]}\] (e.g. go 20000)

If no value is specified for the number of debugging cycles, the debugger runs all the available cycles.

After running a debugging cycle, the debugger is no longer pointing to the beginning of the source code but it is pointing to another source code line. For example after 20000 simulation cycles, the debugger points to the source code line at address [0x00165].

The user can understand how the local memory (LPM) has been mapped on the common memory (CM) at any time of the debugging by using the 'showmap' command.

**5.4. Design Process**

As mentioned in section 1.3 'purpose of the thesis', the goal of this thesis work was to design the snapshot functionality for the simulator system. The snapshot functionality saves the simulator state and retrieves it into the system. So the snapshot functionality consists of two main events, saving the state information, and recovering the saved information back to the simulator system. A block diagram of these events the shown in figure 7.
Saving and retrieving the state information has been designed as the separate commands: 'snapshot' and 'recover'. Figure 8 shows a simple example how these commands to be used through the user interface. In this example the user takes a snapshot of the simulation state after 2000 debugging cycles and then retrieves the saved snapshot state by using the recover command. Internally the 'snapshot' command calls the snapshot function to save all the state information in the snapshot file. The 'recover' command calls the recover function to retrieve the snapshot state back to the debugger. The design steps of these functions are explained in the following.
5.4.1. Take the snapshot

In this part I explain the issues that I faced during designing the snapshot function to take the snapshot of the state.

In the first step I would need a clear description of the snapshot function and explain that what it was going to do. The snapshot function takes a snapshot of the simulation state, so that it takes the state information from the debugger and save it into the snapshot file.

The first question raised was: how many snapshot files are needed? Whether all the state information can be saved in only one snapshot file or a bunch of snapshot files are needed. The second question was what file format is suitable for the snapshot file, text or binary? These questions answered regarding to the purpose of designing the snapshot functionality. Since the snapshot functionality was designed to improve the communication between the developers and the snapshot file must be small in size to be transferred easily between the user stations.
text and binary file formats. Both formats could be used in the snapshot; but a binary file format was preferable for the following reasons:

- It needs less disk space
- It is easier and faster to be transferred from the memory to disk
- The snapshot file is read and retrieved by the processor and does not need to be human-readable

I decided to use only one snapshot file because if there were several snapshot files, some of them may be lost while transferred.

The snapshot file structure was decided but it was empty. What data should be saved in this snapshot file? Where was the state information located in the debugger?

**Memories:** I studied the architecture of the system to find the memory components of the system. As explained in chapter 4, the flexASIC system contains a number of DSPs which used a common memory component, CM (figure 3). In addition, each DSP has its own local program memory (LPM) and local data memory (LDM) (figure 4).

It should be mentioned here that because of the time limitation of this thesis work, I designed the snapshot functionality for one DSP of the system, however since the DSPs have the same architecture, the designed snapshot functionality can be useful for the other DSPs similarly.

In addition to the functions designed for this thesis project, I also used some functions which already existed in the debugger. I have changed the predesigned functions based on the snapshot functionality needs. For example, the snapshot function gets access to the CM, LPM and LDM, collects and saves all the state information into the snapshot file. The flexASIC debugger already used a predesigned function, `flex_get_memory_data_dsp` to get access to the memory component. In this predesigned function, the name memory component should be entered through command line by the user. However, I have changed this function to be used in the snapshot
functionality, as the name of the memory component: CM, LPM and LDM are given internally through a switch case to the debugger.

Registers: Besides of memories, the debugger consists of a number of registers. The values of the registers should be saved in the snapshot file as well as a part of the state information.

The documents about the registers of the system were reviewed; the debugger system has 188 registers with unsigned long integer values. Each register is identified by a specific string name and an integer ID.

To save the register values I designed two new functions, \texttt{int\_showregisters\_id} and \texttt{flex\_get\_register\_by\_id}. In the process of saving the registers’ values, first the snapshot function calls \texttt{int\_showregisters\_id}. Then it internally calls the function \texttt{flex\_get\_register\_by\_id} as many times as the number of registers in order to get all the registers’ id and save the registers’ values in the snapshot file.

Mapping information: Saving the state information from the memory components is necessary but it is not sufficient. The snapshot functionality is saving the state information to be recovered later back to the system; so, the memory mapping information also should be saved in to the snapshot file in order to understand how to address the state information again on the memory components while retrieving the snapshot state.

Object information: As mentioned in section 5.3.4 \textit{‘an example of using the general commands of the debugger’}, the debugger used the 'pgm.obj' file to initialize the registers' values for the testing purpose. Indeed, the obj file can be considered as the source of the simulation state data; since the purpose of snapshot functionality is rolling back to the previous simulation state, the object information should be saved into the snapshot file as well to initialize the registers with the same values as before.
But what information should be saved as the object information? To answer this question I studied the debugger system to find how the debugger loads the object file into the system. The debugger uses a function which is called 'loadobj' that gets the object file name from the user interface as the input. I decided to only save the name of the object file into the snapshot file for reproducing the loadobj function with the same name during the retrieving process. I designed the flex_get_memory_object_file_name function to get the name of the object file from the user interface and save it into the snapshot.

Since the main purpose of using the snapshot functionality is receiving the snapshot state back to the debugger, the state information should be saved as sectioned data to facilitate the snapshot recovering process. To classify the data information, I have considered an identifier in the snapshot file to specify the beginning and end of each memory. Since the size of the memories are large and the exact used size of them is unknown, the snapshot file has been implemented based on dynamic block length.

The sequence of saving the data into the snapshot file is not important; but since the snapshot information should be retrieved in a certain sequence, I decided to save the state information in the same sequence to facilitate the recovering process. As shown in figure 9, the object information should be the first data that is saved and retrieved to the debugger because the debugger requires the object information to initialize the registers' values while the sequence of saving and retrieving the memory components is not so restricted.

<table>
<thead>
<tr>
<th>Snapshot File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Information</td>
</tr>
<tr>
<td>Memory Mapping</td>
</tr>
<tr>
<td>Contents of CM</td>
</tr>
<tr>
<td>Contents of LPM</td>
</tr>
<tr>
<td>Contents of LDM</td>
</tr>
<tr>
<td>Contents of Registers</td>
</tr>
</tbody>
</table>

Figure 9: The sequence of the state information in the snapshot file
5.4.2. Retrieve the Snapshot

After taking the snapshot, the snapshot file consists all snapshot simulation state information and the mapping information. As shown in figure 9, I have designed the ‘recover’ command to retrieve all the saved state information back to the debugger system. When using the ‘recover’ command, the recover function is called by the debugger. In this part, I am going to explain the design process of the recover function.

Object information: The object information is the first component which has been retrieved to the debugger system to initialize the registers' values. As mentioned before, the name of the object file has been saved in the snapshot file. So I developed a procedure to read the object's name from the snapshot file and load the object in to the debugger. To load the object, I used `flex_load_elf_object` function which is an internal predesigned function of the debugger in order to load the object file in to the debugger system. Thus the same object file has been loaded into the debugger.

Mapping information: The mapping information is the second state information which is retrieved from the snapshot file. I developed a procedure to read the mapping lists from the snapshot file and copy them in to the mapping list of the system. The procedure reads the memory mapping information as a node list object from the snapshot file and gives it to the `flex_put_memory_mapping` function to put it as a memory mapping list into the debugger system. Indeed, `flex_put_memory_mapping` reads each node of the list as a new mapping information and set it into the corresponding node of the memory mapping by calling the `em_dsp_put_pgm_mem_map` predesigned internal function.
**Contents of the memories:** The second step is retrieving the contents of the memories which have been saved in the snapshot file. As mentioned in session 5.4.1 “Take snapshot” a number of identifiers were considered in the snapshot file to identify the beginning and end of each memory. In the retrieving snapshot process, I designed a procedure to go through the snapshot file and look for the identifier to find the beginning of the values of each memory which were saved already in the snapshot file. Then the snapshot file is read as much as the size of each memory in a buffer format. This buffer is given to the predesigned function `flex_set_memory_data_dsp` as the input to set the corresponding memory to the buffer value.

**Values of Registers:** To recover the registers’ values, I designed a procedure to look for the identifier to find the location of the registers values in the snapshot file. First the register id and its corresponding value are read as an integer and a buffer from the snapshot file. Then the `flex_set_register_by_id` function is called in order to set the corresponding value into the register. This function is a new function that receives the register id and the value as the inputs and calls the `em_dsp_put_register` internal function to put the register’s value in to debugger system. All steps of the snapshot retrieving process has been shown in figure 10 as a block diagram.
5.5. Example interactions

In this section I will give an example scenario where a user takes a snapshot of the simulator state and retrieves it again into the system. The following steps provide an overview how each component interacts to each other internally.
1. The user wants to take a snapshot of the simulator state so he enters the snapshot command in the command line user interface.

2. The application uses the snapshot module to save the state information into snapshot file.

3. The snapshot module calls the functions `flex_get_memory_data_dsp`, `flex_get_memory_mappings` and `flex_get_memory_object_file_name`\(^1\) to get state information and write them on a snapshot file.

4. After putting all state information into the snapshot file, the debugger gives control back to the user interface.

5. Later, the user wants to retrieve the snapshot simulator state so he enters the recover command in the command line user interface.

6. The application uses the recover module to read the snapshot file and retrieves the snapshot state into the debugger.

7. The snapshot module calls the functions `flex_set_memory_data_dsp`, `flex_put_memory_mapping` and `flex_load_elf_obj`\(^2\) to set the data into their corresponding memory components.

8. After retrieving all the snapshot state information into the debugger, the debugger gives control back to the user interface.

---

\(^1\) The functions explained in part 5.4.1 'Take the snapshot'

\(^2\) The functions explained in part 5.4.2 'Retrieve the snapshot'
Chapter 6. Evaluation

6.1. Overall approach of testing

This section provides the testing process and the results. The testing methods have been designed to cover the goals of the project.

The goals of the project:

- Taking a snapshot of the simulation state and saving the state information into the file.
- Retrieving the snapshot state into the system

Section 5.4 'Example interactions' gives a sample scenario when a user takes a snapshot of the simulator state and a user recovers the snapshot state back to the debugger. I tested this scenario by taking a snapshot of the simulation state and then retrieving the snapshot.

For 20 times I used the snapshot functionality to evaluate the operation effect of the system. Basically, there are two parts I tested in this process. The first part tested saving and retrieving the simulation state. The second part tested the retrieved simulation state in order to continue simulator debugging. Each part follows by its testing results.
6.2. Testing process of saving and retrieving snapshot of the state

The testing process of this part is taking snapshot of the simulation state and retrieve it back to the debugger. As illustrated in figure 11, if the snapshot functionality works properly, the retrieved state should be the same as simulation state snapshot was taken. But how it can be recognized that the save and retrieved simulation state are equal?

Figure 11: Block diagram of using the snapshot functionality for saving and retrieving the state information
In this experiment the address line and the source code line of the debugger compared in two simulation states before taking and after retrieving the snapshot. The debugger points to the same memory address before taking and after retrieving the snapshot.

Figure 12 shows an example of testing taking a snapshot of the simulation state and retrieving it back to the debugger. As shown in this figure, I start the debugger, load the pgm.obj file into the debugger, run the debugger for a number of cycles (e.g. 40000 cycles) and take a snapshot of the simulation state by using snapshot command. I look at the line of the source code and the memory address at this simulation state by using 'showsrc' command and then quit the debugger.

I start the debugger again, retrieve the snapshot by using recover command and look at the memory address and the line of the source code again. As figure 12 shows, the debugger points to the same memory address and the line of the source code is the same as before. This example is a good sign that the snapshot functionality works properly in saving and retrieving the state.
The memory address and the line of the source code before taking the snapshot

The address of the memory and the line of the source code after retrieving the snapshot state

Figure 12: Example of testing saving and retrieving the simulation state after 40000 running cycles
During all 20 tests the snapshot functionality successfully saved and retrieved the snapshot of the state when the numbers of the simulation cycles changed from 1 to the last cycle. The success ratio is 100%. Every time the pointed memory address and the line of the source code at the retrieved state was the same as the snapshot state, as shown in Table 1.

<table>
<thead>
<tr>
<th>Test No.</th>
<th>Number of cycles</th>
<th>Source code before taking snapshot</th>
<th>Source code after retrieving snapshot</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
<td>[0x0000006] (0x0006) ploadr 2</td>
<td>[0x0000006] (0x0006) ploadr 2</td>
</tr>
<tr>
<td>2</td>
<td>3500</td>
<td>[0x000010] (0x0010) ploadr a7</td>
<td>[0x000010] (0x0010) ploadr a7</td>
</tr>
<tr>
<td>3</td>
<td>20000</td>
<td>[0x001675] (0x1675) mv 13879, a1</td>
<td>[0x001675] (0x1675) mv 13879, a1</td>
</tr>
<tr>
<td>4</td>
<td>50000</td>
<td>[0x003cbe] (0x3cbe) orh 46982, a1</td>
<td>[0x003cbe] (0x3cbe) orh 46982, a1</td>
</tr>
</tbody>
</table>

Table 1: Sample tests of testing the snapshot functionality to save and retrieve the snapshot
6.3. Testing process of snapshot functionality to continue simulation

In this part I tested whether the retrieved state can be used for continuing the simulation.

There is a fact that the debugger has the same debugging behavior with the retrieved simulation state and the former state. As shown in figure 13, if I run the debugger for \( N \) number of cycles, the obtained simulation state should be the same as the state that achieved by continuing debugging from the retrieved \( m \)-cycles state for \( (N-m) \) numbers of cycles.

![Figure 13: Block diagram of using the retrieved snapshot for continuing debugging](image)

Figure 13: Block diagram of using the retrieved snapshot for continuing debugging
As shown on the figure 14, in the second experiment first I ran the debugger for all available cycles and took a snapshot of state. I saved this snapshot file, Snapshot File 1 as the scale file.

After that, I start the debugger again and retrieve a snapshot of m number of cycles (e.g. the snapshot of 40000 cycles) and continue debugging for all left available cycles. I take a snapshot from the last simulation state and save as Snapshot File 2.
I compare this snapshot file with the criteria snapshot file which was provided before. I used "differ" Linux command to compare these two snapshot files. This example showed that the snapshot functionality properly using the retrieved snapshot to continue debugging.

20 tests have been conducted in this experiment. I used the snapshot files of different number of cycles, the number of cycles have been selected variously to extend study on the behavior of the debugger during using the retrieved states. In 13 tests the snapshot functionality successfully used the retrieved snapshots to continue simulation; however in 7 tests using the retrieved state were not equal to the criteria state. What are the differences? And why they have been raised?

The difference is at register #184. I reviewed the documents about the registers of the debugger system. There are 188 registers in the debugger system, the registers work in pairs. The registers of the system are involved in a set of computations. So that the values of the registers are changed based on the final results of the computations in each debugging cycle, when the user takes a snapshot of the simulation state, the snapshot function saved all the information which has been already located in the memories and registers on that time. And during retrieving the snapshot, the saved data are set into their corresponding memory component.

First I made a hypothesis that the difference is a constant value which happened just during retrieving the snapshot. To investigate this hypothesis I reviewed the retrieved data in the investigated test cases. The retrieved register’s values were the same as before at the time that the snapshot was taken. So the difference value was happened while using the retrieved state for continuing debugging. Moreover I understood that the difference is not a constant value because in some test cases there is no difference with the criteria value.

I summarized the findings and reviewed the circumstances of the test cases: The retrieved values were the same as before while the snapshot has been taken, the set of the computations were also the same as the criteria test case. Thus the difference may be created before taking the snapshot.

To more investigate on the reason of the difference, I reviewed the architecture of the debugger system again; as figure 4 shows, the state information has been transfer to the memory components through data bus. During taking the snapshot, the values already located into the debugger and can be read from the memory component and were saved into the snapshot. The
results of the second experiment show that in some test cases the final value of the register had not been transferred from the data bus to the memory components yet. These states are called invisible states in the debugger system, so that the values of these states have not got visible yet on the components.

To investigate more on invisible states, I studied the data transferring process in a FlexASIC DSP Core. As shown in figure 4 “Block diagram of a general FlexASIC DSP”, a FlexASIC DSP core was implemented based on instruction pipeline technique which allows the execution of multiple instructions in a time. Pipeline throughput and latency are two main issues in pipeline technology. Pipeline throughput is the measure of how often an instruction exits the pipeline. [37] The pipeline latency is the measure of how long it takes to execute an instruction in the pipeline. [35] [36]

The FlexASIC DSP is divided into two groups; program control instruction and combinations of moves and operations. All instructions execute in one DSP clock cycle unless stated otherwise. When several moves and operations are combined to execute in parallel, the timing will be decided by the instruction that needs the largest number of clock cycles to execute. [10]

The throughput (pipe-lining) and the latency (delay in cycles to availability of result for a dependent instruction) are different for each instruction. For example the latency of the instruction ‘cmp compare register with register’ is fully pipelined and produces a result that is available to an instruction in the subsequent cycle. However the operation ‘div24 24-bit unsigned integer division’ is not fully pipelined, and produces a result that is available to a subsequent instruction after 14 cycles.
Chapter 7. Conclusion and Future Work

1.1. Conclusion

Due to the importance of flexibility for the hardware simulators, a possibility to have a snapshot functionality for the Softsim simulator system was needed. This study focuses on the development of a snapshot functionality for the simulator system. The literature review contains the background knowledge on modeling and simulation, virtual machines and hardware simulators. The state saving methods was studied thoroughly, and the simulator system architecture was reviewed.

Through the development several challenges were raised. Fetching the state information from the simulator system and retrieving the snapshot state back to the debugger were the biggest ones.

The design process of the snapshot state explained in the paper. In addition the paper explored, I) whether the developed snapshot functionality saves and retrieves the state information. II) Whether the retrieved state is useful to continue the simulation. The results of the evaluation show that the snapshot functionality works well for saving and retrieving the simulation state. However the retrieved state was just used properly for continuing simulation visible states.
7.2. Future Work

Still there are some improvements that were not in the scope of this project but could be achieved. A suggestion can be improve the snapshot functionality to take snapshot from the invisible states as well. Another suggestion can be designing and developing a graphical user interface for managing the snapshots of the simulation states. It gives ability to the testers and developers who works with the simulator system to get access to the different versions of snapshots and share them with others simply through the user interface. In addition, the current snapshot functionality is available for one DSP of the debugger and it can be expanded for the multiple DSPs. Moreover a large scale study on the collaboration between the users during applying the snapshot functionality can be conducted to provide them more editing capabilities on the simulator system.
Chapter 8. References


[18] “FlexTool Simulator Overview”, Ericsson AB.


Appendix.

The results of the first experiments, testing the snapshot functionality for saving and receiving the simulation state. The number of simulation cycles has been randomly selected.

<table>
<thead>
<tr>
<th>Test No.</th>
<th>Number of cycles</th>
<th>Source code before taking snapshot</th>
<th>Source code after retrieving snapshot</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
<td>[0x0006] (0x0006) ploadr 2</td>
<td>[0x0006] (0x0006) ploadr 2</td>
</tr>
<tr>
<td>2</td>
<td>3500</td>
<td>[0x000010] (0x0010) ploadr a7</td>
<td>[0x000010] (0x0010) ploadr a7</td>
</tr>
<tr>
<td>3</td>
<td>10000</td>
<td>[0x2bfa] (0x0746) iwait</td>
<td>(0x0746) iwait</td>
</tr>
<tr>
<td>4</td>
<td>20000</td>
<td>[0x001675] (0x1675) mv 13879, a1 l mv 4, a1h</td>
<td>[0x001675] (0x1675) mv 13879, a1 l mv 4, a1h</td>
</tr>
<tr>
<td>5</td>
<td>40000</td>
<td>[0x2bfa] (0x2bfa) dload *a7, *r6++</td>
<td>[0x2bfa] (0x2bfa) dload *a7, *r6++</td>
</tr>
<tr>
<td>6</td>
<td>50000</td>
<td>[0x003cbe] (0x3cbe) orh 46982, a1 l sub a7, a3, a3</td>
<td>mod 16</td>
</tr>
<tr>
<td>7</td>
<td>6000</td>
<td>[0x000010] (0x0010) ploadr a7</td>
<td>[0x000010] (0x0010) ploadr a7</td>
</tr>
<tr>
<td>8</td>
<td>12000</td>
<td>[0x000027] (0x0027) sext a7h</td>
<td>div40 l rol1 a2</td>
</tr>
<tr>
<td>9</td>
<td>15400</td>
<td>[0x2bfa] (0x2bfa) dstore *r500++, *a6</td>
<td>[0x2bfa] (0x2bfa) dstore *r500++, *a6</td>
</tr>
<tr>
<td>10</td>
<td>17420</td>
<td>[0x0011a6] (0x11a6) dstore *r6++, *a5</td>
<td>[0x0011a6] (0x11a6) dstore *r6++, *a5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>11</td>
<td>9600</td>
<td><code>[0x000051] (0x0051) reti .a5:le</code></td>
<td><code>[0x000051] (0x0051) reti .a5:le</code></td>
</tr>
<tr>
<td>12</td>
<td>8300</td>
<td><code>[0x0019d9] (0x19d9) iwait</code></td>
<td><code>[0x0019d9] (0x19d9) iwait</code></td>
</tr>
<tr>
<td>13</td>
<td>37600</td>
<td><code>[0x2bfa](0x2bfa) dload *a3, a2l, .r7:ne</code></td>
<td><code>[0x2bfa](0x2bfa) dload *a3, a2l, .r7:ne</code></td>
</tr>
<tr>
<td>14</td>
<td>8340</td>
<td>`[0x0001e3] (0x01e3) mv 13, nw</td>
<td>add 13, a0`</td>
</tr>
<tr>
<td>15</td>
<td>9500</td>
<td><code>[0x0005b9](0x05b9) pstore .outbox:empty</code></td>
<td><code>[0x0005b9](0x05b9) pstore .outbox:empty</code></td>
</tr>
<tr>
<td>16</td>
<td>10580</td>
<td>`[0x00002b] (0x002b) divs a3 , a4</td>
<td>nop</td>
</tr>
<tr>
<td>17</td>
<td>18900</td>
<td><code>[0x000050] (0x0050) sub 1, a5</code></td>
<td><code>[0x000050] (0x0050) sub 1, a5</code></td>
</tr>
<tr>
<td>18</td>
<td>30400</td>
<td>`[0x002be90] (0x2be90) Crc a0 , a7</td>
<td>min2 a6, a4, a2, .a4:eq, inh`</td>
</tr>
<tr>
<td>19</td>
<td>64700</td>
<td>`[0x000023] (0x0023) Scale2 a5, a0</td>
<td>clb a4l, a2</td>
</tr>
<tr>
<td>20</td>
<td>57000</td>
<td><code>[0x004eec] (0x4eec) mv 0, r0</code></td>
<td><code>[0x004eec] (0x4eec) mv 0, r0</code></td>
</tr>
</tbody>
</table>

**Table 2:** The results of first experiment, saving and retrieving the snapshot of the state