NIISim, a Simulator for Computer Engineering Education

EMIL BÄCKSTRÖM

Master of Science Thesis
Stockholm, Sweden 2012
TRITA-ICT-EX-2012-33
NIISim, a Simulator for Computer Engineering Education

Emil Bäckström
emibac@kth.se

February 2012

Supervisor: Fredrik Lundevall, KTH ICT, flu@kth.se
Examiner: Prof. Mats Brorsson, KTH ICT, matsbror@kth.se
Abstract

Students at KTH can take a course called IS1200 Computer Engineering. This course teaches some of the basic aspects of computer engineering. One important part of the course is the labs which are carried out on an Altera DE2 Development and Educational board. The labs utilize many of the buttons and LEDs on this board. Unfortunately, these boards are only available during the course lab sessions meaning students have no way of fully testing their programs at home. Altera does provide a simulator, but it is not able to simulate the features on the board. NIISim aims to solve this problem.

NIISim (Nios II Simulator) is a simulator that will be able to simulate all the functionality on the DE2 board that is necessary to complete all the IS1200 course labs. It comes with support for the Nios II CPU from Altera, several of Altera’s I/O devices and many features on the DE2 board. With a simple graphical user interface the user is able to quickly load the appropriate files and start the simulation. The user is also able to communicate with the simulated program using a console that supports both text input and output.

Testing has shown that NIISim simulates the IS1200 course labs without problems. This is a great success. Furthermore, the simulation is performed at a much faster rate than the simulator provided by Altera. The intention is now that NIISim will be used in the IS1200 course to help increase students learning experience as they will have much more time to experiment with the DE2 board features. NIISim also makes a great starting platform for future master’s thesis projects such as implementing a cache simulator or multi-core simulation support.
# Table of contents

1. **Introduction** .......................................................................................................................... 5  
   1.1 Background ......................................................................................................................... 5  
   1.2 Motivation .......................................................................................................................... 5  
   1.3 Objectives and requirements .............................................................................................. 6  
      1.3.1 Objectives ...................................................................................................................... 6  
      1.3.2 Requirements ................................................................................................................. 6  

2. **Background study** .................................................................................................................. 8  
   2.1 Existing simulators .............................................................................................................. 8  
      2.1.1 MipsIt ............................................................................................................................ 8  
      2.1.2 SMPCache ..................................................................................................................... 9  
      2.1.3 SimpleScalar ................................................................................................................. 10  
      2.1.4 Spim-Cache ................................................................................................................... 10  
      2.1.5 SpimVista ..................................................................................................................... 11  
   2.2 Cache memories ................................................................................................................... 11  
      2.2.1 Introduction ................................................................................................................... 11  
      2.2.2 Structure and address mapping ..................................................................................... 13  
      2.2.3 Write policies ................................................................................................................ 15  
      2.2.4 Write buffers ............................................................................................................... 15  
   2.3 Cache visualization methods .............................................................................................. 16  
   2.4 DRAM technologies .......................................................................................................... 17  
      2.4.1 DRAM .......................................................................................................................... 17  
      2.4.2 SDRAM, DDR RAM and SGRAM ............................................................................... 19  
   2.5 Altera I/O boards and devices .......................................................................................... 19  

3. **The user interface** ............................................................................................................... 21  
   3.1 Main program ..................................................................................................................... 21  
   3.2 Console windows ............................................................................................................... 22  
   3.3 I/O board window ............................................................................................................. 23  
   3.4 Register window .............................................................................................................. 23  

4. **Implementation** .................................................................................................................... 25  
   4.1 File formats ....................................................................................................................... 25  
      4.1.1 The .sdf file format ...................................................................................................... 25  
      4.1.2 The .board file format ............................................................................................... 30  
      4.1.3 The .elf file format ................................................................................................. 32  
   4.2 Devices .............................................................................................................................. 34  
      4.2.1 Nios II CPU ............................................................................................................... 34  
      4.2.2 SDRAM memory ....................................................................................................... 34  
      4.2.3 UART interface ....................................................................................................... 35  
      4.2.4 JTAG UART interface ............................................................................................. 36  
      4.2.5 LCD interface ........................................................................................................... 37  
      4.2.6 Hardware timer ......................................................................................................... 38  
      4.2.7 PIO interface .......................................................................................................... 39  
   4.3 The Instruction Set Simulator ........................................................................................... 39  
      4.3.1 Nios II ISS ............................................................................................................... 40  
      4.3.2 The ISS in NIISim ..................................................................................................... 40  
   4.4 The I/O board ..................................................................................................................... 42  
      4.4.1 Device groups .......................................................................................................... 43  
      4.4.2 Board devices ........................................................................................................... 43
4.4.3 The LCD display ................................................................. 46
4.5 Threads .................................................................................. 46
  4.5.1 Simulation thread ............................................................. 46
  4.5.2 Update thread ................................................................. 48
4.6 Trace file generation ............................................................. 48
  4.6.1 Dinero file format ............................................................ 49
4.7 Command line parameters .................................................... 49

5. Testing ...................................................................................... 51
  5.1 The IS1200 labs ................................................................. 51
  5.2 Comparison with the Nios II ISS ........................................ 52

6. Conclusions ............................................................................ 55
  6.1 The project objectives and requirements ............................. 55
  6.2 Future work ......................................................................... 56
    6.2.1 Cache simulator .......................................................... 56
    6.2.2 More debugging windows ........................................... 56
    6.2.3 Multi-core simulation .................................................. 56

7. References ............................................................................... 58

Appendix A. Test program .......................................................... 59
Appendix B. Example .sdf file ................................................... 60
Appendix C. Example .board file ............................................... 61
1. Introduction

1.1 Background

Computer engineering is a broad subject. It includes everything from the smallest pipeline stage in a modern CPU to how the main memory, CPU and I/O devices are connected and how they communicate with each other. It also includes low level programming and cache design.

At KTH there is a course that teaches the basic concepts of computer engineering. The name of the course is IS1200 Computer engineering. Labs are an essential part of that course. Most of the labs are carried out on a special board called the Altera DE2 Development and education board. This board is used to represent a basic computer system with a CPU, a main memory and some I/O devices. Programs executing on this board can be written in both low level languages such as assembly and in high level languages such as C++. The environment that is normally used to code and simulate programs in is the Nios II IDE.

The DE2 board has many I/O devices. A few of these are LEDs, toggle switches, an LCD display, serial ports, USB ports, etc. All these devices can be utilized by the programmer. There is, however, a major drawback when writing programs that utilizes I/O devices. These programs cannot be simulated in the Nios II IDE because the program used for simulation, called the Nios II ISS, cannot simulate the I/O devices. It is only able to simulate the instructions that are executed on the Nios II CPU. Due to this drawback, the only way to test if the program is working correctly is to run it on the DE2 board.

Cache design is also a big part of the IS1200 course. To understand how caches work, a special lab has been dedicated for this purpose. In this lab, a simulator called MipsIt is used to simulate the caches. Parameters such as the cache size, block size and associativity can be easily configured to match the specific cache that is of interest. What makes this cache simulator interesting is that it represents the cache as a table together with arrows that shows what data is accessed. Because of this graphical representation, it is very easy to understand how the cache works. However, the programs that were initially written for the DE2 board cannot be used my MipsIt. Students have to write separate programs for the MipsIt cache simulator.

1.2 Motivation

The main idea behind this thesis is to develop a simulator platform that is able to simulate various I/O devices on the Altera DE2 board. Many different I/O devices exist on the DE2 board but not all should be implemented. Only the ones that are used in the computer engineering course labs at KTH are of interest. A few of these are the LEDs, the toggle switches, buttons and timers. The simulator should also not be locked to the DE2 board. It should work with the two scaled down version DE0 and DE1 boards as well.

The purpose of developing this kind of simulator is so that students can test their programs at home before they come to the lab sessions. The Nios II ISS, which is otherwise used by students to test their programs, is not able to simulate all I/O devices. With the new simulator they will be able to load their programs, and in the simulator environment there will be LEDs and buttons that correspond to and work exactly like the LEDs and buttons on the DE2 board. This has the potential of increasing the student’s learning experience.
Another part of this thesis is to also develop a CPU cache simulator. Currently, MipsIt is the program that is used in the computer engineering course to simulate caches. It is an old simulator and it is not compatible with the Nios II programs, so the purpose here is to develop a new one that works with the programs written for the DE2 board. This will be convenient for students as they’ve already familiarized themselves with the Altera DE2 board and the Nios II IDE. They won’t have to switch to a different simulator platform such as MipsIt.

Finally, if there is enough time to spare, a pipeline simulator should also be developed. The idea here is to create a 5-stage pipeline simulator that is similar to the one that exists in MipsIt simulator collection. This could be useful for future courses that focus more on CPU pipelines.

Good knowledge of the Altera DE2 board and its I/O devices, the Nios II instruction set, caches and graphical windows programming will be required to develop this simulator. All this together with the desire that it should increase student’s learning experience makes it suitable to be a master’s thesis project.

1.3 Objectives and requirements

1.3.1 Objectives
Here is a list of the main objectives of this thesis.

- Develop a simulator platform that can simulate the Altera DE0, DE1 and DE2 boards and CPU caches.
- Make sure all labs, apart from the ones that involve the serial port, in the IS1200 course run without bugs in the simulator.
- Make sure that the cache simulator can be used to complete the cache lab in IS1200.

If these three objectives have been completed and there is enough spare time left, a fourth objective will be introduced.

- Develop and integrate a 5-stage pipeline simulator into the simulator platform.

1.3.2 Requirements
Some main requirements that the simulator platform should satisfy are listed down below.

- The simulator should be written in C++ and compiled for Microsoft Windows.
- It should be designed so that it can run under Linux systems using Wine.
- It should be very easy to add a pipeline simulator in the future. This is only valid if there is not enough time to add one during the design phase.
- The simulator should be designed in such a way that it is easy to extend it to simulate systems with multiple Nios II processors.
- The I/O devices that are to be simulated are only those that are used in the IS1200 labs. No other I/O devices should be implemented in the simulator.
- The cache simulator should be very configurable. One should be able to set basic cache parameters such as cache size, block size, associativity, replacement policies, write policies and size of write buffers. There should also be an option for specifying both read and write access times to the main DRAM memory. An example of how this
could be specified is on the form 3-1-1-1, meaning a burst transfer where it takes 3 bus cycles to complete the first transfer and 1 bus cycle for the three subsequent transfers.

- The cache simulator should be able to support large cache sizes. This will require scrollbars and zooming functions to make it easy to visualize them for the user.
- The cache simulator should support a method for visualizing where hits and misses occur in the cache.
- To be able to see where hits and misses occur in the cache when the program is running, a slide-bar is needed to vary (slow down) the simulation speed. Otherwise it is very likely that the cache will update colours too fast.
- In the cache simulator there should be an option to simulate to the main() function in the program. This could be useful if the user wants to skip the Altera HAL initialization code.
2. Background study

2.1 Existing simulators

Simulators are important tools when designing and analyzing computer systems. For instance, using a cache simulator can answer the question what the optimal cache configuration is to run a particular program. Once that answer is known, a chip can be manufactured with those specific cache parameters used to solve the problem. Many different simulators exist in the world of computer engineering. I will take a closer look at five of them here and see how well they meet the requirements of NIISim.

2.1.1 MipsIt

MipsIt was developed at the Lund University in Sweden [1]. It is a collection of simulators as well as a developing environment tool. The developing environment was inspired by Microsoft Visual Studio and serves as a tool for developing software for a specific hardware platform. The same software can also be simulated by the MipsIt simulator collection. The hardware platform is an IDT development board that is in use at Lund University. The development board contains a micro controlled with a MIPS32 ISA processor.

The simulators that come with MipsIt include a system simulator for the IDT board, a cache simulator as well as a pipeline simulator. Figure 2.1 shows the system and cache simulator main window.

![Figure 2.1 MipsIt system and cache simulator](image)

In addition to this window where we can see the system architecture, it is possible to open more windows that show the contents of the RAM, the registers in the CPU, a console for text output and controls for the I/O devices. The user can load a program into the RAM and then start the simulation by pressing the green “play” button. The CPU will then start executing the code located in the RAM. Pressing the D-Cache box or the I-Cache box will bring up a new
window showing the contents of each cache. If the simulation is running, the user will see a real time view of the cache as it’s being accessed. That is, the user will see how the contents in the cache update and how each memory access in the cache is handled. This is done by using animation. Some cache statistics are also displayed such as hit count, miss count, hit rate and cycle count.

MipsIt has a very good way of visualizing the cache contents using a 2D grid method. However, when it comes to large caches or caches with high associativity, the simulator fails to visualize it properly. The user will not be able to see the entire cache. Some parts of the grid will be outside the screen and thus won’t be visible to the user. There is no way to scroll the cache vertically and horizontally. One of the requirements of NIISim is that it should be able to visualize large caches. The MipsIt cache simulator clearly does not satisfy that requirement.

It is possible to configure several important cache parameters in MipsIt. They include cache size, block size (words per block), associativity, replacement policies, write policies, memory access times and the size of write buffers. All these parameters are included in the requirements for NIISim. But the way access times are specified in MipsIt does not satisfy our requirements. In MipsIt all memory accesses take the same number of cycles, and it is this number of cycles the user can select. It is not possible to specify access times on the form 3-1-1-1.

MipsIt also incorporates a pipeline simulator. The pipeline simulator was developed to be very flexible. Instead of having a fixed pipeline architecture, the user can load custom made pipeline architectures. These pipeline architectures are written in a HDL language developed specifically for the pipeline simulator.

2.1.2 SMPCache

SMPCache is a 1 level cache simulator developed for symmetric multiprocessor (SMP) systems [2]. It was developed at the University of Extremadura in Spain. An SMP system is a system consisting of two or more identical CPUs that share the same main memory. Since the CPUs are identical, they also have identical caches. This brings up the concept of cache coherence since the caches in each individual CPU have to communicate with each other to be coherent. While this is not something that will be supported in NIISim, it is something of great importance if NIISim is extended to support multi-core simulation in the future.

One of the reasons why SMPCache was developed was because a multi-core simulator called bigDIRN, which was used at that time to simulate caches in multi-core systems, had some limitations such as portability (bigDIRN is a simulator for UNIX only), no graphical interface and little analysis of result parameters. Much focus was put on the analysis part so that it could be used for not only educational purposes, but for research purposes as well. This required the simulator to be very configurable and produce a lot of output data in various forms.

The graphical interface of SMPCache lets the user configure both the caches and the main memory in a variety of different ways. Some of the cache parameters that can be configured are number of cache lines, number of words per cache line, associativity, replacement policies and the size of a word. It is also possible to select different cache coherence protocols. MSI, MESI and DRAGON are the three cache coherence protocols the user can choose from.
As mentioned before, SMPCache produces a lot of output data. After simulating a test program, the user can analyze miss rates, hit rates, number of memory accesses, number of state transitions in each cache and much more. It is also possible to view graphs describing how for example the miss rate depends on a cache variable. State transition diagrams are provided for each cache as well.

### 2.1.3 SimpleScalar

SimpleScalar is a collection of simulators for processors and caches [3]. They include functional simulation, profiling, cache simulation and out-of-order simulation. All simulators are execution-driven, they are heavily optimized for performance and they can be run on many host systems including Windows and Linux. The only requirement for running the tools is that the GNU tools are installed. All simulators are command line based programs. They have no graphical user interface and the output data is in text format only.

The binary files the user wants to simulate using the SimpleScalar tools must be compiled using a modified version of the GNU GCC compiler. This compiler is provided with the simulators. The reason why a special compiler is used is because SimpleScalar only accepts binaries with a certain ISA called the SimpleScalar ISA. Since SimpleScalar uses its own ISA it makes sense to use a special compiler for it. The SimpleScalar ISA is similar to MIPS but with some modifications that makes simulation easier to perform.

The functional simulator is called **sim-fast**. It is optimized for raw speed and does no time accounting, instruction checking and uses no caches. The cache simulators are called **sim-cache** and **sim-cheetah**. The difference between the two is that **sim-cheetah** uses a very efficient method to generate simulation results for fully associative caches. They both support L1 and L2 cache simulations. The cache parameters that can be set are number of cache lines, block size, associativity and replacement policy. Write policies cannot be selected. Therefore the cache simulators do not meet the requirements for NIISim.

The profiling simulator is called **sim-profile**. It supports several different profiling features such as instruction class profiling, instruction profiling, branch class profiling and many more. The final simulator called **sim-outorder** is the out-of-order simulator. It is the most complicated one. It simulates the execution of instructions in a processor that has an out-of-order execution pipeline. A huge number of parameters exist to allow the user to customize the pipeline architecture in many different ways.

### 2.1.4 Spim-Cache

Spim-Cache is a processor and cache simulator. It was developed at the Polytechnic University of Valencia in Spain [4]. Unlike other cache simulators that are normally trace-driven, Spim-Cache is execution-driven. It extends another simulator called Spim which runs MIPS32 assembly language programs. The MIPS32 instruction set was chosen because it is widely used in course books. The Windows version of Spim is called PCSpim.

PCSpim consists of a window that is split in four horizontal sections containing the register file, the program code, the memory contents and a log window. Spim-Cache adds two additional seconds to this window containing the instruction cache and the data cache. It is possible to customize the caches in several ways. The user can select cache size, block size, associativity, write policies and replacement algorithms in the cache settings window. However, the user is forced to choose from a set number of cache sizes and block sizes
(namely four different cache sizes and three different block sizes). This is a limitation that should not be present in NIISim. The simulation of the caches work by intercepting load and store instructions as they are executed by PCSpim.

### 2.1.5 SpimVista

SpimVista is another cache simulator [5]. It is similar to Spim-Cache in the way that it extends the Spim simulator. It was developed at the Technical University of Valencia in Spain. Since it is extends Spim, it is an execution-driven simulator as well.

SpimVista was developed due to a few drawbacks that were found in Spim-Cache. For instance, Spim-Cache is only able to simulate one cache level meaning it can’t be used for simulating systems with cache hierarchies. Multi level cache configurations are very common these days and different cache levels often have different parameters. Thus, SpimVista was developed to tackle this limitation of Spim-Cache. The goal was to improve learning experience for multi level cache memories when using the simulator for educational purposes.

The new cache settings window in SpimVista has improved a lot from the one in Spim-Cache. The user is now able to choose from a much larger selection of cache sizes, block sizes, associativity ways and replacement policies. Different parameters can be selected for the L1 and L2 caches. A large focus has also been put in visualizing the caches. There is a new cache window where the user can view the contents of both the L1 and L2 caches. The caches are represented as 2D matrices where the rows represent the cache lines and the columns represent the ways in the cache. This means the cells in the matrix represent a cache block. The simulator also visualizes hits and misses by colourizing cells in the matrices. Hits are visualized by a green colour and misses by a red colour. The actual data in the caches can be viewed by moving the mouse over the cells.

Another feature of SpimVista is the ability to roll back execution. This is useful when studying the LRU replacement algorithm. Sometimes one might not realise which block got replaced. Then rolling back the execution by one clock cycle will reveal that particular block.

A downside with SpimVista is that it has a hard time visualizing large caches. The simulator is not able to display all rows in such cases. To deal with this, SpimVista hides rows that are not currently being used. Once they are used they will be displayed again. Since SpimVista fails to visualize large caches it does not meet one of the requirements of NIISim.

### 2.2 Cache memories

#### 2.2.1 Introduction

Cache memories [7], [8] are used to increase the performance of the system. The largest bottleneck is usually the main memory. It takes a lot more time for the CPU to access the main memory than it takes to access a register. This is because when accessing the main memory, it has to go through the system bus. Also, the main memory usually consists of DRAM cells which are slow compared to SRAM and CPU registers.

To cope with this bottleneck, cache memories are used. A cache memory is normally made up or SRAM cells and is placed between the CPU and the bus. It intercepts all memory references made by the CPU and checks if the data is in the cache. If it is, it immediately returns it to the CPU. This takes a lot less time than having to go through the bus to access the DRAM main memory. This is illustrated in figure 2.2.
Figure 2.2 A cache and a non-cache system

Cache memories take advantage of two important principles: Principle of temporal locality and principle of spatial locality. The principle of temporal locality says that if a memory location was referenced at one point in time, it is very likely to be referenced again in the near future. Examples of this are a counter variable that is continuously updated and a program loop where the same instructions are executed over and over again in a short period of time. The principle of spatial locality says that if a memory location was referenced at one point in time, it is very likely that near memory locations will be referenced in the near future. Examples of this are sequential data in an array and sequential program execution.

Because computer programs use these principles often, we can make the cache memory small compared to the main memory but still keep a very high hit rate. Hit rate and miss rate are two important properties of a cache memory. Hit rate tells us how often we find the data in the cache. If we have done 100 memory references and for 80 of those we found the data in the cache, the hit rate is 80%. The miss rate is the opposite of the hit rate. In the previous example the miss rate is 20%.

For CPUs, the cache memory usually comes in two variants: an instruction cache and a data cache. The instruction cache holds the data that encodes the instructions that are to be executed by the CPU. The data cache holds all other data that is referenced by the program. Since these two caches have two different purposes, they normally have different parameters as well. The most common parameters are cache size, block size and associativity. They will be explained later on. Caches can also come in different levels. For example, if the data is not found in the data cache, we go to a “level 2” cache. If the data is not found in the level 2 cache, only then we go to the main memory. This is illustrated in figure 2.3. The level 2 cache in figure 2.3 is a unified level 2 cache. Some systems also have level 3 caches.

The following sections will describe how a cache memory is built up. Since it is much smaller than the main memory there will obviously be memory collisions since all addresses in the main memory does not have a unique place in the cache. This is handled by address mapping.
2.2.2 Structure and address mapping
Caches consist of a number of cache blocks, also called cache lines. Common sizes for a cache block are 8 bytes, 16 bytes, 32 bytes, etc. The size of a cache block should be divisible by the word size, which we will assume to be always 4 byte. The number of cache blocks in the cache is determined by the total cache size and the size of each block. If the cache has a size of 16 kb and each block has a size of 32 bytes, there are \((16*1024) / 32 = 512\) blocks in the cache.

The cache blocks hold the 4 byte data words together with a tag. The tag is used to identify which main memory address the data at this particular block correspond to. Each block also has a valid bit that indicates whether the data in this block is valid or not. Figure 2.4 shows an example cache with 4 cache blocks, each having a size of 16 bytes.

When data are transferred from the main memory to the cache, an entire line is transferred even though only one data word was accessed. This is because we take advantage of the principle of spatial locality. If we are accessing the word at address 0x00001000, we add the words at addresses 0x00001004, 0x00001008, 0x0000100C to the cache as well since it is very likely that they will be accessed too.
We need to figure out which cache line the data at main memory address 0x00001000 is going to be placed in. The address mapping is what takes care of that. The main memory address will be divided into several parts. Those parts will tell us where the data should be put in the cache. Figure 2.5 shows a 32-bit address divided into three parts: a tag, a line and an index. The bit range for these parts depends on the cache properties. In figure 2.5 it is assumed that we have a cache with a block size of 16 bytes and a total of 4 blocks.

![Figure 2.5 Address mapping](image)

We see that bits 0 and 1 are not used. This is because we have assumed that the word size is 4 bytes. Hence we only accept memory addresses that are divisible by 4. All such addresses will have bits 0 and 1 set to zero. The offset part, bits 2 and 3, specify in which of the four slots in the block the data word will be placed in. The index part, bits 4 and 5, specify in which of the four blocks the word will be placed in. Finally we have the tag consisting of bits 6 to 31. The tag is used when we search for data in the cache. Note that if the block size were 32 bytes, the offset would be 3 bits long. That is because we then have 8 slots in each block, hence we need 3 bits to cover all those 8 slots. The index would still be 2 bits long size we still have only 4 blocks. The tag however would be one bit shorter.

The cache that has been discussed so far is called a direct mapped cache. That is because each memory address maps to exactly one specific line in the cache, determined by the index and offset bits. Another type of cache is a set associative cache. In a set associative cache, the memory address maps to two or more different cache lines. If the memory address maps to two different cache lines, the cache is said to be 2-way set associative. A 2-way set associative cache can be thought of as two direct mapped caches placed next to each other. Figure 2.6 shows an example of such a cache. In figure 2.6 the total cache size is 128 bytes. Each cache block is 16 bytes wide. That gives us a total of 8 cache lines. These 8 cache lines are split between two direct mapped caches. We then have two direct mapped caches each constituting of 4 cache lines.

A special case of a set associative cache is when the memory address can map to as many places as there are cache lines. Such a cache is called a fully associative cache. Each direct mapped cache would then only have 1 line.
Since we have a total of 8 cache lines, one would assume that we need to use 3 index bits from the main memory address to point out the cache line. That is not the case here. We only use 2 index bits since each direct mapped cache only has 4 lines. A question that then immediately arises is how do we select which direct mapped cache the memory address should be mapped to? The answer is that we use what is called a replacement policy. Common replacement policies are random, FIFO (First In First Out) and LRU (Least Recently Used).

### 2.2.3 Write policies

The write policy affects the cache behaviour on write hits. The write policy can be either write through, or write back. In write through, the data is written to both the cache and the main memory. In write back, the data is only written to the cache and only when that cache line is replaced is the data written back to the main memory. This requires a dirty bit for each cache line. The dirty bit indicates if the cache line holds a modified (dirty) copy of the data, or if the data is the same as the data in the main memory.

Just like there are two ways of handling write hits, there are two ways of handling write misses. The behaviour of the cache when a write miss occurs is controlled by its write allocate setting. If write allocate is enabled, the cache first retrieves the data from the main memory and then treats the write miss as a write hit meaning what happens after that is controlled by the write policy. If write allocate is disabled, the cache is simply ignored and the data is written directly to the main memory.

### 2.2.4 Write buffers

When it comes to updating the main memory with data from the cache, the CPU is often stalled until the write transfer on the bus is completed. This happens if the write policy is set to write through and a write hit occurs. The CPU updates both the cache and the main memory. CPU stalls are never wanted. A common way to reduce these stalls is to implement write buffers. Whenever the cache sees that the main memory needs to be updated, it puts the data along with the address in a write buffer. The write buffer then takes care of updating the main memory by issuing write transfers on the bus while the CPU can continue executing the next instruction.
2.3 Cache visualization methods

This section is about visualization. More specifically how to visualize how caches affects the execution of a program. Many cache simulators discussed in section 2.2 produce output data such as hit rate, miss rate, hit count, miss count and cycle count. This is one way of telling the user how well the program is executing. If the miss rate is very high, then the user knows that something probably can be done to improve the performance. Similarly if the cycle count dropped significantly after adding a cache, and the miss rate is acceptable, the user knows that the performance has increased due to the introduction of the cache.

However, we’d like to visualize this information in a better way than just as numbers in a table. MipsIt and SpimVista are two simulators that have focused on this. Both simulators use a 2D grid to visualize the cache contents. MipsIt then uses animation to show the user how the cache contents update when hits and misses occur. SpimVista takes this one step further by colorizing each cell in the 2D grid with distinct colours for hits and misses. By doing this the user will be able to look directly at the cache to see if there is a lot of hits and misses. This also helps understanding how the cache works and why misses happens.

There are also other ways of visualizing how caches affect program execution. Instead of showing the cache contents, one can track each memory access in the cache (hits and misses) and display them in a 2D frame [6]. Since there will most likely be lots of memory access we can represent each memory access as one pixel starting at the top left corner in the 2D frame. Its colour will depend on if it was a hit or miss. Now as we keep executing the program, more and more memory accesses occur and we keep adding pixels to the frame (representing the memory accesses) next to the previous one until we hit the end of the frame. Then we move on to the next like and start filling that one with pixels. When the execution of the program is complete we will get something that looks like figure 2.7.

![Figure 2.7 Cache miss pattern](image)

In figure 2.7 black pixels represent cold misses, dark grey pixels represent capacity misses, light grey pixels represent conflict misses and white pixels represent hits. We can clearly see...
the intensity of misses here as there are many black/grey pixels. It is also possible to spot various patterns in the frame.

2.4 DRAM technologies

2.4.1 DRAM

In today’s PCs, the main memory is usually made up of DRAM chips. DRAM stands for dynamic random access memory [9]. Random means that it takes a fixed amount of time to retrieve the data regardless of its address. The word dynamic means that DRAM memory chips need to be refreshed dynamically in order to keep their data valid. The data is stored as charges in small capacitors and since capacitors lose their charge over time, they need to be refreshed in order to preserve the data. Figure 2.8 shows a simplified version of a typical DRAM chip.

![Figure 2.8 DRAM chip](image)

The data is stored in the memory cell array as individual bits. The bits are arranged in a square matrix so that they can be addressed by a row- and column address. If the memory cell array has a size of 16 Mb, that is 16,777,216 bits, it has 4096 rows and columns. To minimize the number of address bits needed to address one of the 16,777,216 bits, two address are driven to the DRAM chip in succession. The first address is the row address and the second one is the column address. This means we only need 12 bits to address 4096 rows and columns instead of a 24 bit address that specifies both the row and column address at the same time.

The two signals RAS (row address strobe) and CAS (column address strobe) are used to determine if the address that is supplied is a row or a column address. Usually the row address is supplied before the column address. This means that the so-called RAS – CAS delay is an important variable of a DRAM chip. It tells us how long we must wait before we can supply the column address to the chip after we’ve supplied the row address. When both the row and
column addresses have been decoded, the write enable signal (WE) controls if we are going perform a write or read operation. A read operation causes the addressed bit in the memory cell array to be transferred to the I/O gate and then later to the data out signal. If instead we are doing a write operation, the data in bit is transferred to the I/O gate and then later written to the memory cell array.

A read or write operation is done by taking the following steps:

1. The RAS signal is driven and the row address is supplied simultaneously.
2. The CAS signal is driven and the column address is supplied simultaneously.
3. Depending on the state of the WE signal we either write the data from the data in pin to the memory array, or we read the data from the memory array and output it on the data out pin.
4. Stop driving the RAS and CAS signals when the operation is complete

There are three different methods to refresh the capacitors that hold the data in the memory cell array. The first method is called RAS-only refresh. In RAS-only refresh we perform a dummy read operation. Whenever a read or write operation is being executed, all bits located at the row specified by the row address is automatically refreshed when the row address is decoded. This means we can drive the RAS signal and supply the address of the row we want to refresh and then skip driving the CAS signal. We simply stop driving the RAS signal when the refresh is complete. After that we can perform another dummy read operation on the next row until the entire memory array is refreshed. A drawback of using this method is that it requires some external logic, or a software program, that loops through all rows and refreshes them. That is why this method is not used that often.

The second refreshing method is called CAS-before-RAS refresh. In this method the DRAM chip has its own internal refreshing logic with an address counter. The refreshing logic is activated when the CAS signal is driven during a certain amount of time while the RAS signal is not. The row specified by the address counter is then refreshed and the address counter counts up by one so that the next time a refresh is activated, the next row will be automatically refreshed.

The third method is called hidden refresh. Hidden refresh is similar to CAS-before-RAS. The refresh is said to be hidden because it happens directly after a read operation. Normally both the RAS and CAS signals stop being driven when the read is complete. In hidden refresh the CAS signal is still active so that a CAS-before-RAS refresh is automatically triggered in the next cycle. There is an internal address counter here as well that counts up and selects the row that is to be refreshed. CAS-before-RAS and hidden refresh are the most used refreshing methods.

DRAMs have several operating modes to increase performance and deliver the data back faster. The read or write operation described earlier is called a normal mode operation. Two other modes are page mode and hyper page mode (EDO mode). Page mode tries to take advantage of the fact that if read or write operations are being done on addresses with the same row address (but different column address) there is no need to decode the row address again. So to read four bits in succession in page mode one first performs a normal read but keeps the RAS signal driven after the first read is complete. Then the CAS signal is switched and a new column address is supplied to select a different column. After switching the CAS signal two more times, four bits will have been read in a smaller time span than if all four reads were done in normal mode. In hyper page mode the time distance between two
consecutive CAS switches is shorter than in page mode. This further reduces the average access time of the DRAM chip.

### 2.4.2 SDRAM, DDR RAM and SGRAM

SDRAM is an evolution from DRAM. It stands for Synchronous Dynamic RAM. SDRAMs have a clock signal as opposed to normal DRAMs. This opens up for more advanced operations and pipelined control logic. SDRAMs work in burst mode which is similar to DRAM EDO mode. In burst mode the first memory transfer takes a few more clock cycles than the subsequent memory transfers. This is because there is no need to decode the row address for the subsequent memory transfers just like in DRAM EDO mode. Burst transfers are usually specified on the form 3-1-1-1 meaning the first transfer took 3 clock cycles and the following three transfers only took 1 clock cycle each.

DDR RAM stands for Double Data rate DRAM. They can double the rate of data transfers by transferring data not only on the rising edges of the clock, but also on the falling edges. There is also a type of DRAM called SGRAM (synchronous graphic RAM) that is optimized for graphics cards. They work the same as SDRAMs but are optimized for the fastest possible data transfer instead of highest possible memory capacity.

### 2.5 Altera I/O boards and devices

Altera provides a number of different development and educational boards. Three of them are the DE0, DE1 and DE2 boards. It is the Altera DE2 board that is used in the IS1200 course. The other two boards are basically just scaled down versions of the DE2 board. The main component on these boards is an FPGA. FPGA stands for Field-Programmable Gate Array. It is an integrated circuit that can be programmed to model a variety of digital hardware systems. In our case, we are interested in digital hardware systems that consist of a Nios II CPU, SDRAM memories and I/O devices, all connected on the same bus. All these components are designed by Altera.

The systems are created using a tool called Altera SOPC (System On a Programmable Chip) builder. In the Altera SOPC builder it is possible to design and customize system architectures in a variety of different ways. When a system architecture has been created, the SOPC builder generates VHDL code that matches the system. The VHDL code can then be compiled and the result is a file which can be used to program the FPGA on one of the I/O boards.

Here is a list of all devices that NIISim needs to be able to simulate:

- Nios II /s CPU
- SDRAM memory
- UART interface
- JTAG UART interface
- LCD interface
- Hardware timer
- PIO interface

All these devices can be added to a system using the SOPC builder.
The Nios II CPU is a 32-bit processor. It comes in three different types: Fast, standard and economy. The standard type is the one that is used in the IS1200 course. Therefore it will be the CPU to focus on.

The SDRAM memory works like any normal computer memory. On the I/O boards, the SDRAM is accessed by first going through an SDRAM controller. However, this controller does not need to be modelled because we are not interested in what happens on the RTL level. We are only interested in the functionality of the SDRAM memory. Thus, it will be sufficient to simply model the SDRAM controller as the SDRAM memory itself.

UART stands for Universal Asynchronous Receiver/Transmitter. On the DE2 board, a UART interface is used for serial communication using the RS-232 connector.

JTAG stands for Joint Test Action Group. A JTAG interface is normally used for debugging. In our case, it is used in conjunction with a UART interface which allows for communication between the board and the PC using a console window. This sort of communication is possible with the normal UART interface as well, but using the JTAG UART interface it is possible to send an entire text string at one instead of individual characters.

The LCD interface is connected to the physical LCD on the DE2 board. It enables programs to communicate with it and print out text.

Hardware timers are devices that have an internal counter that counts down from a specific value, called period, to zero. They can be used for measuring or simulating time.

The PIO interfaces are usually connected to the buttons and the LEDs on the board. PIO stands for Parallel Input Output. It allows a program to turn on and off LEDs and read button states.
3. The user interface

The idea behind the user interface of NIISim is to give the user as much freedom as possible. NIISim has several child windows for viewing consoles and controlling the I/O board. All of them are hidden by default. In most cases, the user won’t be needing all of them when working with the simulator. Therefore, the user is able to show only the child windows that satisfies his requirements. The user is also able to freely move and all of the windows across the entire screen. This allows him to customize the simulation environment in a way that he is most comfortable with.

3.1 Main program

The main program consists of a small window that has a menu bar and various toolbar buttons that incorporates most of the important features of NIISim. It is shown in figure 3.1.

![NIISim main program](image)

The functionality of each button is explained in table 3.1 starting with the left most button and going right.

<table>
<thead>
<tr>
<th>Button</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load program</td>
<td>Loads a program file (.elf)</td>
</tr>
<tr>
<td>Load system description file</td>
<td>Loads a system description file (.sdf).</td>
</tr>
<tr>
<td>Run</td>
<td>Starts the simulation</td>
</tr>
<tr>
<td>Single step</td>
<td>Executes one instruction only and then pauses the simulation</td>
</tr>
<tr>
<td>Pause</td>
<td>Pauses the simulation</td>
</tr>
<tr>
<td>Stop</td>
<td>Stops the simulation</td>
</tr>
<tr>
<td>I/O board</td>
<td>Shows/hides the I/O board window</td>
</tr>
<tr>
<td>Consoles</td>
<td>A dropdown button where the user can select which console windows to show/hide.</td>
</tr>
<tr>
<td>Registers</td>
<td>Shows/hides the registers window</td>
</tr>
<tr>
<td>Generate trace file</td>
<td>Starts or stops generating a trace file</td>
</tr>
</tbody>
</table>

Table 3.1 Button functionality

All the functionality of the buttons can be accessed using the menu bar as well. In the CPU menu bar there is an additional way of starting the simulation. This option is called “Run (slow)”. It will start the simulation in slow mode, executing instructions much slower than in the normal (fast) mode. Notice that the pause and stop buttons are greyed out in Figure 3.1. This means that the simulation is already stopped. Thus it can’t be paused or stopped again so these two buttons are disabled (greyed out). When the user starts the simulation, the pause and stop buttons will be enabled and the run button will be disabled. This makes it easy for the user to control the simulation. Having all four buttons that control the simulation enabled at all times would be confusing.
Before the simulation can be started the user must load a system description file and a program file. The system description file must be loaded before the program file. The other way around would not work since NIISim has to know the base addresses of all SDRAM memories in the system before it can load a program file. The base addresses of the SDRAM memories are located in the system description file. If the user attempts to load a program file first, an error message will appear.

When trying to load a new system description file or a new program file while the simulation is running, NIISim will inform the user that the current simulation must be stopped before a new file can be loaded. NIISim will ask the user if he wants to proceed loading the new file or not. If the user chooses to load a new file, NIISim will automatically stop the current simulation.

3.2 Console windows

NIISim comes with three different console windows. One that can be connected to a JTAG UART interface and two that can be connected to standard UART interfaces. The JTAG UART console window is simply called “JTAG UART” and the two UART console windows are called UART0 and UART1. Figure 3.2 shows the JTAG UART console window with some text output. The UART0 and UART1 consoles have the same layout.

![Figure 3.2 JTAG UART console window](image)

The console works like a normal terminal console. It is resizable. It is possible to type in text directly into the text window. Pressing enter will send that text to the program. It is also possible to select a portion of the text and copy it just like in a normal text editor. This makes it easy for the user to for example copy text and paste it into a lab report document. There is a Clear button which will empty the text area.

The difference between the JTAG UART console and the UART0 and UART1 consoles is the way the characters that are being typed in are sent to the simulated program. In the JTAG UART console it is possible to type in multiple characters. It is not until the user presses enter here that the text is sent to the program. In the case with the UART0 and UART1 consoles, the characters typed in are immediately sent to the program. There is no need for the user to press enter here. It is still possible to press enter in those consoles but that will just bring the cursor down to the next line.
3.3 I/O board window

The I/O board window is very customizable. NIISim is not locked to a fixed board. Instead, the board is described in a .board file. This file is loaded when the user loads a system description file. When no board is loaded, this window just shows the text “No I/O board has been loaded”. Figure 3.3 shows the board window after the user has loaded a system description file. The board here is the Altera DE2 board. It is the board that is used in the IS1200 labs.

![Figure 3.3 Altera DE2 board](image)

The layout of this board is completely described in the .board file. The .board file format is described in section 4.1.2. We can see how this board resembles the real Altera DE2 board. It should be easy for a user who is familiar with the Altera DE2 board to recognize the buttons and LEDs.

The initial size of the window is determined by the size of the background image which in this case is the dark blue rectangle. Resizing the window is possible if the user wants to work with another size. When the window becomes smaller than the size of the background image, horizontal and vertical scrollbars will appear so that it is possible to view the entire board even when the window is small. In the other situation when the window becomes larger than the background image, the board will be positioned in the center of the window.

The user can interact with the board by pressing the toggle switches and the push buttons. That is possible even when the simulation is not running. When the simulation is running and the program is set to turn on some LEDs, the LEDs will immediately light up on the board. A program can also write text to the LCD window in the top left corner.

3.4 Register window

The register window will show the contents of all the registers in the CPU, including control registers as well as the program counter. If the system description file contains more than one
CPU (which is possible), the register window will only show the contents of the registers in the CPU that was first added to the system.

An example of the register window when simulating a program is shown in figure 3.4.

![Figure 3.4 Register window](image)

The register window is resizable just like all other child windows. It consists of a table with 4 columns and 19 rows. Columns 1 and 3 list the registers and columns 2 and 4 list the values in the registers in hexadecimal form. The register names for the 32 general purpose registers are shown as r0-r31. Registers which have specific identifiers linked to them are shown with them as well. An example of this is r0 / zero. This makes it easy for the user to find these registers as the identifiers are often used in assembly language instead of the standard names.

The contents of four control registers are shown in the bottom rows. They are identified by the names status, estatus, ienable and ipending. Their respective control register numbers are 0, 1, 3 and 4. Finally at the bottom, the value of the program counter is displayed.

Double-clicking on a register value will bring up a new window that allows the user to change the value of the register that was clicked. The values of all registers except the pc can be changed. The number the user has to input when changing a value must be a hexadecimal number. This number has to be entered without prefixes and suffixes. An example would be if one is to type the hex number 0x0000FA00, one would type 0000FA00 (not 0x0000FA00 or 0000FA00h). Here the first four zeros can be omitted. It would be enough to just type FA00.

It is only possible to change the registers while the simulation is either running or paused. That is because when the user starts the simulation from a stopped state (not paused state), the simulator will issue a CPU reset which will bring the values of all registers back to 0.
4. Implementation

4.1 File formats

There are several different file formats to be aware of when using NIISim. First we have the .sdf file, or system description file. This file has information about the system architecture. More specifically, it has information about how many SDRAM memories, hardware timers, UART interfaces, etc are present in the system. It also specifies their respective memory mappings, IRQ values and component specific parameters. The .sdf file is also able to load a .board file (this file format is explained below) and specify how the devices on the board should be mapped to any PIO interfaces in the system.

The information about the system architecture is actually already present in two other files that are used when you want to run your programs real physical board. One might ask, why create a new file format when there are already two existing files that has the information needed? There are two reasons for this. First, one of the files, called the .sof file, is the file that has information about how all the logical elements should be connected together on the FPGA on the board. In a sense, this means it does describe the system. However it describes it on an abstraction level (RTL level) which is far too low for our needs. The second file, called the .ptf file, is the file used when creating programs that are to be run on the board. It has information about all the components in the system, but it lacks information about what components are mapped to what devices on the board. Also, the .ptf file has a very complex structure and contains a lot of other information we don’t need. Therefore I decided to create a very simple file format that only has information about what components the system consists of and how they are mapped to the devices on the I/O board.

The second file format used by NIISim is the .board file. This file describes the I/O board. NIISim is not locked to the DE2 Board that is used in the IS1200 course. Instead, the I/O board can be a customized which makes the simulator compatible with the Altera DE0 and DE1 boards as well. The .board file can’t be loaded manually like the .sdf file. This is because if you want to switch I/O boards, the components that were mapped to devices on the previous board might not be compatible with the new board. Also, the new board might have different names for the onboard devices so you most likely need to set up new mappings which will require a load of a new .sdf file. Because of this connection between the .sdf file and the .board file, the .board file is loaded by a specific command in the .sdf file.

The last file is the .elf file. This is a standard file format for compiled programs that contain executable code. NIISim only accepts .elf files which are compiled for the Nios II processor. Attempting to load a file compiled for any other processor will fail. The .elf file contains the actual program to be simulated. Information about where the executable code should be located in memory is contained in this file. Therefore an .sdf file must be loaded before an .elf file, otherwise NIISim will be unable to copy the executable code into a SDRAM memory.

4.1.1 The .sdf file format

The .sdf file is a plain text file. It consists of several commands, each specifying what component to add to the system. There is also a command to load a .board file and a command to map components to devices on the I/O board. Comments can be added in the file to make it more readable. Comments begin with the standard C++ comment symbol // and ends as the end of the line. It is not possible to use the C commenting style /* … */ to place multi line comments. Table 4.1 lists all valid commands and a short description of them.
AddCPU

Syntax: AddCPU Name, Reset address, Exception address, Frequency

This command adds a Nios II CPU core to the system. It is possible to add more than one CPU, but only the first CPU that is added will be able to run the program. It is also only possible to view the registers of the CPU that was added first. Table 4.2 lists the description of all the parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>A name which identifies the CPU.</td>
</tr>
<tr>
<td>Reset address</td>
<td>Integer</td>
<td>The address from where the CPU should start executing code after a reset.</td>
</tr>
<tr>
<td>Exception address</td>
<td>Integer</td>
<td>The address from where the CPU should start executing code after a software exception.</td>
</tr>
<tr>
<td>Frequency</td>
<td>Integer</td>
<td>The frequency of the CPU in Hz. On the DE2 Board this is equivalent to the system clock since the CPU is modelled with a CPI value of 1.</td>
</tr>
</tbody>
</table>

AddSDRAM

Syntax: AddSDRAM Name, Base address, Span

This command adds an SDRAM memory to the system. It is the only type of memory that can be added. SRAM and flash memories are not supported. It is possible to add several SDRAM
memories mapped to different areas in the system address space. Table 4.3 lists the description of all the parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>A name which identifies the SDRAM.</td>
</tr>
<tr>
<td>Base address</td>
<td>Integer</td>
<td>The base address of the SDRAM. This will be the address in the system address space which the SDRAM is mapped to.</td>
</tr>
<tr>
<td>Span</td>
<td>Integer</td>
<td>The chunk of memory (in bytes) that is going to be mapped to the SDRAM, starting at the base address.</td>
</tr>
</tbody>
</table>

Table 4.3 AddSDRAM command parameters

AddUART

Syntax: AddUART Name, Base address, Span, [IRQ]

This command adds a UART (Universal Asynchronous Receiver/Transmitter) interface to the system. It is used for serial communication. A UART interface can be mapped to one of the two UART consoles. It is possible to add several UART interfaces mapped to different areas in the system address space. Table 4.4 lists the description of all the parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>A name which identifies the UART interface.</td>
</tr>
<tr>
<td>Base address</td>
<td>Integer</td>
<td>The base address of the UART interface. This will be the address in the system address space which the UART interface is mapped to.</td>
</tr>
<tr>
<td>Span</td>
<td>Integer</td>
<td>The chunk of memory (in bytes) that is going to be mapped to the UART interface, starting at the base address.</td>
</tr>
<tr>
<td>IRQ</td>
<td>Integer</td>
<td>The interrupt request (IRQ) number. This parameter is optional.</td>
</tr>
</tbody>
</table>

Table 4.4 AddUART command parameters

AddJTAG

Syntax: AddJTAG Name, Base address, Span, [IRQ]

This command adds a JTAG (Joint Test Action Group) UART interface to the system. The JTAG UART interface is similar to the standard UART interface. It is used to let the physical I/O board communicate with a PC using a console. A JTAG UART interface can be mapped to the JTAG UART console. It is possible to add several JTAG UART interfaces mapped to different areas in the system address space. Table 4.5 lists the description of all the parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>A name which identifies the JTAG UART interface.</td>
</tr>
<tr>
<td>Base address</td>
<td>Integer</td>
<td>The base address of the JTAG UART interface. This will be the address in the system address space which the JTAG UART interface is mapped to.</td>
</tr>
<tr>
<td>Span</td>
<td>Integer</td>
<td>The chunk of memory (in bytes) that is going to be mapped to the JTAG UART interface, starting at the base address.</td>
</tr>
</tbody>
</table>
**IRQ**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ</td>
<td>Integer</td>
<td>The interrupt request (IRQ) number. This parameter is optional.</td>
</tr>
</tbody>
</table>

Table 4.5 AddJTAG command parameters

**AddLCD**

Syntax: AddLCD Name, Base address, Span

This command adds an LCD interface to the system. An LCD interface can be mapped to an I/O board LCD display. It is possible to add several LCD interfaces mapped to different areas in the system address space. Table 4.6 lists the description of all the parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>A name which identifies the LCD interface.</td>
</tr>
<tr>
<td>Base address</td>
<td>Integer</td>
<td>The base address of the LCD interface. This will be the address in the system address space which the LCD interface is mapped to.</td>
</tr>
<tr>
<td>Span</td>
<td>Integer</td>
<td>The chunk of memory (in bytes) that is going to be mapped to the LCD interface, starting at the base address.</td>
</tr>
</tbody>
</table>

Table 4.6 AddLCD command parameters

**AddTimer**

Syntax: AddTimer Name, Base address, Span, Frequency, Period, Period unit, Fixed period, Always run, Has snapshot [IRQ]

This command adds a hardware timer to the system. It is possible to add several hardware timers mapped to different areas in the system address space. Table 4.7 lists the description of all the parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>A name which identifies the hardware timer.</td>
</tr>
<tr>
<td>Base address</td>
<td>Integer</td>
<td>The base address of the hardware timer. This will be the address in the system address space which the hardware timer is mapped to.</td>
</tr>
<tr>
<td>Span</td>
<td>Integer</td>
<td>The chunk of memory (in bytes) that is going to be mapped to the hardware timer, starting at the base address.</td>
</tr>
<tr>
<td>Frequency</td>
<td>Integer</td>
<td>This value should be set to the same frequency as the CPU. It is used to calculate the internal period in clock cycles.</td>
</tr>
<tr>
<td>Period</td>
<td>Integer</td>
<td>The period of the hardware timer. The unit of the period is specified by the next parameter.</td>
</tr>
<tr>
<td>Period unit</td>
<td>String</td>
<td>The unit of the period. Can be either “ms” for milliseconds or “us” for microseconds.</td>
</tr>
<tr>
<td>Fixed period</td>
<td>Integer</td>
<td>This parameter can be either 0 or 1. A 1 means the period of the hardware timer can’t be changed. A 0 means the period can be changed.</td>
</tr>
<tr>
<td>Always run</td>
<td>Integer</td>
<td>This parameter can be either 0 or 1. A 1 means the hardware timer is always running. Once started it can’t be stopped. A 0 means it can be stopped.</td>
</tr>
<tr>
<td>Has snapshot</td>
<td>Integer</td>
<td>This parameter can be either 0 or 1. A 1 means it is possible to take a snapshot of the internal counter. A 0</td>
</tr>
</tbody>
</table>
AddPIO

Syntax: AddPIO Name, Base address, Span, Type, [IRQ]

This command adds a PIO (Parallel Input/Output) interface to the system. A PIO interface can be mapped to various I/O board devices such as LEDs, seven segment displays, push buttons and toggle switches. It is possible to add several PIO interfaces mapped to different areas in the system address space. Table 4.8 lists the description of all the parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>A name which identifies the PIO interface.</td>
</tr>
<tr>
<td>Base address</td>
<td>Integer</td>
<td>The base address of the PIO interface. This will be the address in the system address space which the PIO interface is mapped to.</td>
</tr>
<tr>
<td>Span</td>
<td>Integer</td>
<td>The chunk of memory (in bytes) that is going to be mapped to the PIO interface, starting at the base address.</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
<td>The type of the PIO interface. It can be either “in” for input or “out” for output.</td>
</tr>
<tr>
<td>IRQ</td>
<td>Integer</td>
<td>The interrupt request (IRQ) number. This parameter is optional.</td>
</tr>
</tbody>
</table>

Table 4.8 AddPIO command parameters

ImportBoard

Syntax: ImportBoard Filename

This command loads a .board file. The .board file to load is specified by the parameter Filename. It can be either a full path or a path relative to the NIISim executable program. The path must include the name of the .board file with the .board file extension.

Map

Syntax: Map Component identifier, Device name

This command maps a system component to consoles and I/O board devices. The Map commands should always appear last in the .sdf file since the system components as well as the I/O board have to be loaded before a mapping can be done. Table 4.9 lists the description of all the parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component identifier</td>
<td>String</td>
<td>The identifier of the component that is going to be mapped to a console or I/O board device.</td>
</tr>
<tr>
<td>Device name</td>
<td>Integer</td>
<td>The name of the device which the system component should be mapped to. For the UART consoles it can be either “UART0” or “UART1”. For the JTAG UART</td>
</tr>
</tbody>
</table>
console it can be “JTAG”. For I/O board devices the
device name should match the name of a board device
group.

Table 4.9 Map command parameters

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetName</td>
<td>Assigns a name to the board.</td>
</tr>
<tr>
<td>SetBackgroundImage</td>
<td>Specifies a background image for the board.</td>
</tr>
<tr>
<td>AddDeviceGroup</td>
<td>Adds a new device group.</td>
</tr>
<tr>
<td>AddDevice</td>
<td>Adds a new device.</td>
</tr>
<tr>
<td>AddLCD</td>
<td>Adds an LCD device.</td>
</tr>
</tbody>
</table>

Table 4.10 .board file commands

4.1.2 The .board file format

The .board file has the same format as the .sdf file. That is, it is a plain text file with
commands that are sufficient enough to model the DE0, DE1 and DE2 boards. Modelling
boards consists of creating what I call device groups and board devices. In short, a device
group is something that can be mapped to a PIO interface and board devices are the actual
buttons and LEDs. The concept of device groups and board devices are explained in more
detail in section 4.4. Comments can be added in the file in the same was as in the .sdf file.
Table 4.10 lists all valid commands and a short description of them.

The following section will describe each command in more detail. Parameters are written in
italic form and optional parameters are placed in square brackets [ ].

SetName

Syntax: SetName Name

This command assigns a name to the board. It will show up in the title bar of the I/O board
window. The name is specified by the parameter Name.

SetBackgroundImage

Syntax: SetBackgroundImage Filename

This command specifies a background image for the board. This image must be a .bmp file
which is a standard file format for bitmap files in windows. The .bmp file to load is specified
by the parameter Filename. It can be either a full path or a path relative to the NIISim
executable program. The path must include the name of the .bmp file with the .bmp file
extension.

AddDeviceGroup

Syntax: AddDeviceGroup Name, Type

This command adds a device group to the board. A device group is something that can be
mapped to a PIO interface. The type of the device group must match the type of the PIO
interface. It is possible to add several device groups to the board. Table 4.11 lists the description of all the parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>A name which identifies the device group. It is used when adding board devices to the device group and when mapping the device group to a PIO interface.</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
<td>The type of the device group. It can be either “in” or “out” and it must match the type of the PIO interface it is going to be mapped to.</td>
</tr>
</tbody>
</table>

Table 4.11 AddDeviceGroup command parameters

**AddDevice**

Syntax: AddDevice Type, Device group, Bit, Bitmap filename, X-coordinate, Y-coordinate

This command adds a board device to a device group. A board device can be toggle switches, push buttons, LEDs or seven segment displays. It is possible to add several board devices to the same device group, provided that they are mapped to different bits in the PIO interface register. Table 4.12 lists the description of all the parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>String</td>
<td>The type of the board device. It can be one of the following name identifiers: “LED” for LEDs, “SSLED” for seven segment displays, “PUSH” for push buttons and “TOGGLE” for toggle switches.</td>
</tr>
<tr>
<td>Device group</td>
<td>String</td>
<td>The name of the device group this board device is going to be placed in.</td>
</tr>
<tr>
<td>Bit</td>
<td>Integer</td>
<td>The bit in the PIO interface register this device group is going to be mapped to.</td>
</tr>
<tr>
<td>Bitmap filename</td>
<td>String</td>
<td>An image file that will visualize this board device. This image must be a .bmp file. It can be either a full path or a path relative to the NIISim executable program.</td>
</tr>
<tr>
<td>X-coordinate</td>
<td>Integer</td>
<td>The X-coordinate of the position of the board device image. This coordinate is relative to the top-left corner of the background image in the I/O board window and it increases rightwards.</td>
</tr>
<tr>
<td>Y-coordinate</td>
<td>Integer</td>
<td>The Y-coordinate of the position of the board device image. This coordinate is relative to the top-left corner of the background image in the I/O board window and it increases downwards.</td>
</tr>
</tbody>
</table>

Table 4.12 AddDevice command parameters

**AddLCD**

Syntax: AddLCD Name, X-coordinate, Y-coordinate

This command adds an LCD display to the board. It is not connected to any device groups. Only one LCD is supported on the I/O board. Table 4.13 lists the description of all the parameters.
### 4.1.3 The .elf file format

The .elf file format is a standard format for executable files. ELF stands for Executable and Linking Format [11]. An .elf file is the final output when compiling programs for the Nios II CPU. Thus, understanding of this file is important if one wants to understand how it is loaded.

There are several different types of .elf files. We will focus on the type called “executable file” which holds a program that can be executed. The content of such an .elf file is divided into headers and sections. Figure 4.1 describes the outline of an example .elf file.

![Figure 4.1 ELF outline](image)

The main header contains general information about the .elf file. Such information is a magic number that tells the program that this file is actually an .elf file. Other information is the type of .elf file, file version, endianess, machine code and number of program headers and section headers.

A valid .elf file in NIISim must be of type “executable file”. The file version must be set to EV_CURRENT which is a constant corresponding to the value 1. The endianess must be little-endian and the machine code must be Nios II machine code. If all these conditions are met, NIISim continues to parse the program headers. Otherwise NIISim will display an error telling the user that the .elf file cannot be loaded.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>The name of the LCD display. It is used when mapping the LCD to an LCD interface.</td>
</tr>
<tr>
<td>X-coordinate</td>
<td>Integer</td>
<td>The X-coordinate of the position of the LCD. This coordinate is relative to the top-left corner of the background image in the I/O board window and it increases rightwards.</td>
</tr>
<tr>
<td>Y-coordinate</td>
<td>Integer</td>
<td>The Y-coordinate of the position of the LCD. This coordinate is relative to the top-left corner of the background image in the I/O board window and it increases downwards.</td>
</tr>
</tbody>
</table>
The program header table is an array of headers where each header describes a segment. A segment is a chunk of data that could be either machine code, global variables or other hard coded data. The program header holds information such as the offset of the segment in the .elf file, the size of the segment and the type of the segment. It also holds the address to which the segment should be copied to. This address is normally an address to which a SDRAM memory is mapped to.

NIISim loops through all the program headers and copies each segment from the .elf file to an SDRAM memory. It finds the correct SDRAM memory by looking at the segment address. If this address is not mapped to an SDRAM memory, NIISim displays an error and the .elf file will not be loaded. The copying of segments to SDRAM memories is visualized in Figure 4.2.

![Figure 4.2 ELF file segment copying](image)

Here, we see an .elf file with two segments, one which are to be copied to address 0x00800000 and another one which are to be copied to address 0x00900000. Both are of size 0x00010000. We also have an SDRAM memory (not shown in the figure) mapped at address 0x00800000 of length 0x800000 (8 Mb). Therefore, the SDRAM memory occupies addresses 0x00800000 to 0x00FFFFFF. Both segments fit in this range and they will be copied without errors.

If instead the SDRAM memory had a size of only 0x100000 (1 Mb), the SDRAM would span the address range 0x00800000 to 0x008FFFFF. Segment 2, which is supposed to be copied to address 0x00900000, cannot be copied to our SDRAM memory. NIISim will show an error and abort the .elf loading process. An error will also be shown if the segment has a size that is larger than the SDRAM memory. Note that in this example we have assumed that there is only one SDRAM memory in the system.

When all segments have been copied, the .elf loading process is complete. The machine code of the program and other relevant data now exists in some system SDRAM memories and it is possible to begin execution.
4.2 Devices

This section will discuss the devices that can be simulated in NIISim. They were briefly touched upon in section 2.5. Here they will be explained more detailed. Focus lies on how they are implemented in NIISim, not how they work on the FPGA. The interested reader is directed to [12].

There are 7 devices that NIISim is able to simulate. Out of those devices the Nios II CPU can be viewed as the master in the system because it is the only device that can initiate read and write memory transfers. The remaining six devices (which are all I/O devices except the SDRAM memory) can be viewed as slaves because they can’t initiate memory transfers. Instead, they respond to them. Figure 4.3 shows an example system with a Nios II CPU, a SDRAM memory and two PIO interfaces.

![Figure 4.3 An example system](image)

Note that all devices are connected on the same bus. This important to know and it is true for all devices in the system. Any devices that are added will be connected on the same bus. NIISim does not support multiple buses or bus bridges.

All devices are implemented as classes to encapsulate their functionality. This makes it easy to add several devices of the same type. A new class object is simply instantiated every time a new device is added.

4.2.1 Nios II CPU

As explained in 4.2, the Nios II CPU can be viewed as the master device in the system. It is the only device that can initiate memory transfers. The CPU can’t respond to any memory transfers because it is not mapped anywhere in the system address space. Its functionality is that on every clock cycle, it executes a function called `OnClock()` which simulates one instruction. This is called an ISS (Instruction Set Simulator) and it is explained in section 4.3.

4.2.2 SDRAM memory

The SDRAM memory is a device that models a simple computer memory. It is basically just an array of bytes with the same length as the real SDRAM memory. It is the simplest one of all devices.

One special feature of the SDRAM memory device is that it is possible to directly load it with a large chunk of data from an .elf file without having to call multiple memory writes. This is a
feature that is only used by the simulator itself. It cannot be used by the simulated system. When copying the segment data to the data array, the segment data is not only copied to the data array. It is also copied to a “backup” location. The backup location will hold an exact copy of the segment data and it which will not change during simulation, contrary to the data in the data array which will change all the time as the program writes to the SDRAM memory.

When a system reset is issued, the data in the data array has to be restored to its original state. Instead of reopening the .elf file every time the user stops and restarts the simulation, the SDRAM memory device copies the data from the backup location to the data array. The backup location is simply used to hold a copy of the segment data so that the .elf file does not have to be reopened.

### 4.2.3 UART interface

The UART interface is used for serial communication. It can be mapped to a UART console which will allow the user to view transmitted characters and also input. If the UART interface is not mapped to a console, nothing will happen when a program attempts to transmit data, and it will never be possible to retrieve any data.

The UART interface is a memory mapped device. It has four memory mapped registers. They are explained in table 4.14. The offset is the offset in bytes from the base address of the interface.

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Access type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RxData</td>
<td>0x00</td>
<td>Read</td>
<td>Bits 0-7 holds the data that have been received. The other bits are unused.</td>
</tr>
<tr>
<td>TxData</td>
<td>0x04</td>
<td>Write</td>
<td>Bits 0-7 holds the data that are to be transmitted. The other bits are unused.</td>
</tr>
<tr>
<td>Status</td>
<td>0x08</td>
<td>Read</td>
<td>Bit 6 (TRDY) determines if data can be written to TxData. Bit 7 (RRDY) determines if there is data to be read from RxData. The other bits are unused.</td>
</tr>
<tr>
<td>Control</td>
<td>0x0C</td>
<td>Read/Write</td>
<td>Bit 6 (ITRDY) enables interrupts when data can be written to TxData. Bit 7 (IRRDY) enables interrupts there is data to be read from RxData. The other bits are unused.</td>
</tr>
</tbody>
</table>

Table 4.14 UART interface registers

Note that transfer rates are not simulated. Whenever a character is written to the TxData register, the character is immediately sent to the console (if there is a console mapped to the interface). Therefore, the TRDY bit will always be 1.

If the UART interface has been created with anIRQ number, it can issue read and write interrupts. These two interrupts are controlled by the ITRDY and IRRDY bits. When the ITRDY bit is one and there is new data to be read from RxData, an interrupt is issued. The same thing will happen if the ITRDY bit is one and new data can be written to TxData. Since
data can always be written to TxData (the TRDY bit is always one), a write interrupt will always be pending when the ITRDY bit is one. A read interrupt is acknowledged by reading from RxData and a write interrupt is acknowledged by writing to TxData.

At a reset, ITRDY and IRRDY are set to 0 so that no interrupts will be issued until the simulated software enables them.

### 4.2.4 JTAG UART interface

The JTAG UART interface is similar to the UART interface in the way that it is used for serial communication. It is a UART interface too so it basically sends data in the same way. The difference lies in its interface to the software. It is intended to be used for communicating with a command prompt by sending strings instead of individual characters. For example, if the user were to type in a string in a console that is mapped to a UART interface, the software would have to send each individual character to the UART interface immediately after they are typed in. Using the JTAG UART interface, the software can wait until the user presses the return key. Only then does the text need to be sent and the text can be sent as an entire string, not individual characters.

The JTAG UART interface is a memory mapped device. It has two memory mapped registers. They are explained in table 4.15. The offset is the offset in bytes from the base address of the interface.

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Access type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>0x00</td>
<td>Read/Write</td>
<td>Bits 0-7 holds the data that are going to be send or read. Bit 15 indicates if the data is valid. Bits 16-31 holds the number of characters left in the read FIFO (first in first out) queue. The other bits are unused.</td>
</tr>
<tr>
<td>Control</td>
<td>0x04</td>
<td>Read/Write</td>
<td>Bit 0 (RE) enables interrupts when there is data to be read. Bit 1 (WE) enables interrupts when there is data to be written. Bit 8 (RI) indicates if there is a read interrupt pending. Bit 9 (WI) indicates if there is a write interrupt pending. Bit 10 (AC) indicates activity. Bits 16-31 holds the number of spaces left in the write FIFO queue.</td>
</tr>
</tbody>
</table>

Table 4.15 JTAG UART interface registers

Since transfer rates are not simulated, the write FIFO will always be full of empty spaces. In NIISim it is locked to 64 spaces. The read FIFO also has a limit of 64 characters. However, if one types in a string that contains more than 64 characters, the JTAG UART interface will still accept that string but when reading the number of characters left in the read FIFO, it will be capped at 64. It will stay at 64 until there are less than 64 characters left. At that point it will start to decrease. It has been implemented this way to cope with transfer rates which are present on the real board.
The RE and WE bits control whether read and write interrupts should be issued. If the user types in a string in the JTAG UART console and presses return, that string is sent to the JTAG UART interface and if the RE bit is set to 1, a read interrupt is issued and the RI bit is set to 1. The read interrupt is acknowledged by reading from the data register. When the software enables write interrupts by setting WE to 1, a write interrupt is immediately issued (since the write FIFO is always empty) and the WI bit is set to 1. This indicates that it is possible to send data by writing to the data register. The write interrupt is acknowledged by writing to the data register.

The activity bit AC is set to 1 whenever data is read from or written to the data register. It is cleared by writing a 1 to bit 10 in the control register.

At a reset, RE, WE, RI, WI and AC are all set to 0.

4.2.5 LCD interface
The LCD interface is used to communicate with an LCD display that can be added to the I/O board. The LCD display is of type 2x16 meaning it has 2 lines, each which can display 16 characters. Thus a total of 32 characters can be displayed.

The LCD interface is a memory mapped device. It has two memory mapped registers. They are explained in table 4.16. The offset is the offset in bytes from the base address of the interface.

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Access type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>0x00</td>
<td>Write</td>
<td>Sends an instruction to the LCD interface.</td>
</tr>
<tr>
<td>Data</td>
<td>0x04</td>
<td>Write</td>
<td>Sends data (characters) to the LCD interface.</td>
</tr>
</tbody>
</table>

Table 4.16 LCD interface registers

As can be seen from the table, it is not possible to read any data from the LCD interface.

Valid instructions that can be written to the instruction register are:

- 0x01 – Clears all text on the display.
- 0x80 – Moves the cursor to the top line, starting at the leftmost character.
- 0xC0 – Moves the cursor to the bottom line, starting at the leftmost character.

The real LCD interface has many more instructions which have not been implemented in NIISim. Only the most basic once that support full text output on both lines have been implemented.

When one wants to write characters to the LCD display, one writes them as ASCII characters to the data register. The characters will appear at the position of the cursor which is initially found at the leftmost position in the top line. After the last character in the top line has been written, the cursor moves to the leftmost position in the bottom line. Then it moves back up to the top line when the bottom line is full.

At a reset, the text is cleared and the cursor is positioned to the leftmost position in the top line.
4.2.6 Hardware timer

The hardware timer is a device that can be used for timing. It has an internal counter that can be set to count down from a specific value, called period, to zero. It always counts down by one each clock cycle. Therefore, its class has an `OnClock()` function, just like the CPU class.

The hardware timer also has a few parameters that can only be set at creation time, that is when they are added to the system from the .sdf file. These parameters are: Fixed period, Always run and Has Snapshot.

When the fixed period parameter is set, the period cannot be changed by the software. The always run parameter makes it impossible to stop the timer once started. If the has snapshot parameter is set, it is possible to take a snapshot of the internal counter. Taking a snapshot and reading it is the only way knowing what value the internal counter has.

The hardware timer is a memory mapped device. It has six memory mapped registers. They are explained in table 4.17. The offset is the offset in bytes from the base address of the interface.

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Access type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>0x00</td>
<td>Read/Write</td>
<td>Bit 0 (TO) indicates if there is a timeout (counter has hit zero).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 1 (RUN) indicates if the timer is running.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The other bits are unused.</td>
</tr>
<tr>
<td>Control</td>
<td>0x04</td>
<td>Read/Write</td>
<td>Bit 0 (ITO) enables interrupts when there is a timeout (counter has hit zero).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 1 (CONT) indicates if the timer is in continuous mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 2 (START) starts the timer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 3 (STOP) stops the timer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The other bits are unused.</td>
</tr>
<tr>
<td>Period_low</td>
<td>0x08</td>
<td>Read/Write</td>
<td>Holds the lower two bytes of the period.</td>
</tr>
<tr>
<td>Period_high</td>
<td>0x0C</td>
<td>Read/Write</td>
<td>Holds the upper two bytes of the period.</td>
</tr>
<tr>
<td>Snapshot_low</td>
<td>0x10</td>
<td>Read/Write</td>
<td>Holds the lower two bytes of the snapshot.</td>
</tr>
<tr>
<td>Snapshot_high</td>
<td>0x14</td>
<td>Read/Write</td>
<td>Holds the upper two bytes of the snapshot.</td>
</tr>
</tbody>
</table>

Table 4.17 Hardware timer registers

The TO bit is automatically set to 1 when the internal counter hits zero. It stays at 1 until cleared by software even though the timer might have restarted. It is cleared by writing any value to the status register. The RUN bit is automatically set to 1 one when the timer starts. It stays at 1 until the timer stops. It cannot be changed in between.

To start the timer, one simply writes a 1 to the START bit in the control register. The timer can be stopped at any time (unless the parameter Always run is set) by writing a 1 to STOP. If the CONT bit is set and the internal counter hits zero, it will automatically be reloaded with the value stored in the period register and start counting down again. Setting the ITO bit to 1 causes an interrupt to be issued whenever the internal counter hits zero. The interrupt is acknowledged by writing a 0 to ITO or by writing any value to the status register.
To take a snapshot of the timer, one first has to write any value to one of the two snapshot registers. After that it is possible to read the value.

At a reset, TO, RUN, ITO and CONT are all set to 0. The snapshot register is also reset to 0. The period is loaded with its initial value that was specified at creation time. The internal counter is also loaded with the initial value of the period, but the timer won’t start counting down until the software writes a 1 to the START bit.

4.2.7 PIO interface

The PIO interface is used to allow software to communicate with LEDs and buttons on the I/O board. A PIO interface has 32 I/O ports that can be connected to any device. In most cases these I/O ports are connected to LEDs and buttons. The 32 I/O ports each correspond to one of the 32 bits in the data register. I/O port n corresponds to bit n. The ports can be either input ports or output ports. Which one depends on the parameter in the .sdf file. Bidirectional ports are not supported.

The PIO interface is a memory mapped device. It has three memory mapped registers. They are explained in table 4.18. The offset is the offset in bytes from the base address of the interface.

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Access type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>0x00</td>
<td>Read/Write</td>
<td>This is a 32-bit register in which each bit corresponds to one of the 32 I/O ports.</td>
</tr>
<tr>
<td>Interrupt mask</td>
<td>0x084</td>
<td>Read/Write</td>
<td>Each bit in this 32-bit register controls whether the corresponding port is able to issue an interrupt.</td>
</tr>
<tr>
<td>Edge capture</td>
<td>0x0C</td>
<td>Read/Write</td>
<td>The bits in this register are set to 1 whenever the corresponding port changes value.</td>
</tr>
</tbody>
</table>

Table 4.18 PIO interface registers

As explained, the data register is directly mapped to the 32 I/O ports. If the ports are output ports and the software writes a 1 to bit 5 in the data register, I/O port 5 will be driven with a value of 1. If the ports are input ports and the software reads from the data register, the bits in the data register will correspond to the value of all the I/O ports.

If a bit in the interrupt mask register is set, an interrupt will be issued when that corresponding I/O port changes value. This is only valid for input ports. Output ports cannot issue interrupts. Interrupts are acknowledged by writing any value to the edge capture register. The edge capture register can be used to tell which of the I/O ports have changed values. A bit is set to 1 in the edge capture register whenever that corresponding I/O port changes value from either 0 to 1 or 1 to 0.

At a reset, the data, the interrupt mask and the edge capture registers are all set to 0. This disables interrupts for all I/O ports so that the software can decide which one of them should be allowed to issue interrupts.

4.3 The Instruction Set Simulator

Computer programs stored in the main memory consists of machine code. The machine code is basically assembly instructions that have been translated into 0:s and 1:s. It is the CPU’s
duty to decode and execute those instructions. This is how programs are executed by the hardware and the CPU can be seen as the “core” of the system.

In NIISim we also need to decode and execute machine code, but we can’t use the PC hardware for this. The hardware is already executing NIISim. To be able to execute the machine code of the program that is going to be simulated, we need an ISS (Instruction Set Simulator). An ISS is a simulator that simulates the behaviour of a CPU. This includes simulating the instruction set and handling exceptions and interrupts correctly. The ISS can be seen as the “core” of NIISim just like the CPU is the “core” in the hardware system. The CPU used in the IS1200 labs is the Nios II/s core so this will be our target CPU for the ISS. Our ISS will have to simulate the Nios II instruction set and handle exceptions (both software and hardware) the same way as they are handled by the Nios II/s core.

4.3.1 Nios II ISS

Fortunately for us Altera already provides an ISS that satisfies these requirements, namely the Nios II ISS that is used in the Nios II IDE. Nios II ISS is a command line program and runs in the background. It even supports the simulation of timers which is one of the I/O devices that is used in the IS1200 labs. It would seem like a good idea to use this ISS in NIISim. It would save a lot of time since no new ISS has to be developed. However, there are a few complications. The Nios II ISS does not support stepping through instructions one by one. It executes the entire program from start to finish. There is no way to pause the simulation. Pausing and also slowing down the simulation speed is something that we need to control in NIISim. Furthermore, to implement a cache simulator and memory mapped I/O devices we need to intercept load and store instructions in the ISS, and neither this is possible with the Nios II ISS. These problems could be solved by modifying the source code for the Nios II ISS. Unfortunately though, Altera does not give out the source code.

There is still one possibility that we might be able to use the Nios II ISS. When debugging programs in the Nios II IDE it is possible to step through the instructions one by one. The debugger is also a command line program that runs in the background, just like the Nios II ISS. The debugger is documented very briefly in Altera’s manuals. While it is possible to step through instructions, it is still not possible to intercept load and store instructions. We are therefore forced to conclude that the Nios II ISS cannot be used in NIISim. It simply doesn’t meet our requirements.

4.3.2 The ISS in NIISim

The ISS I have developed for NIISim mimics the functionality of a Nios II CPU, specifically the Nios II /s core [10]. What this means is that NIISim’s ISS and the Nios II /s core have same instruction set architecture and they handle exceptions and interrupts in the same way. Now, the Nios II /s core has features such as 5-stage pipelining, branch prediction, a JTAG DEBUG module and custom instructions. These features have not been implemented in the ISS. The reason why not is simply because we are only interested in executing the instructions. We do not care about how the instructions are executed, that is if they are executed in a pipelined manner. For simplicity all instructions take 1 cycle to execute.

The ISS implements the functionality of all the 32 general purpose registers in the Nios II instruction set. That means register r0 always returns zero when read and writing to r0 has no effect. Register r29 is updated with the address the program should return to after an exception whenever an exception is issued and register r31 is updated with the address the
program should return to after a call instruction is executed. A program can read all registers and write to all registers except for r0.

The ISS also has 4 control registers implemented. There are many more control registers in the actual Nios II architecture but these extra control registers deal with advanced concepts such as memory management units and memory protection units. Neither of those two concepts are implemented in the ISS. The 4 control registers that are implemented are status, estatus, ienable and ipending. Register status has one bit of interest. This is bit 0. It controls whether the ISS should respond to hardware interrupts or not. A value of 1 indicates that hardware interrupts will be handled and a value of 0 turns them off. All remaining bits in the status register have no function. The estatus register holds a copy of the status register when the exception handler is being executed. The ienable register controls which of the 32 IRQ signals are able to issue an interrupt. Each bit in ienable corresponds to one of those 32 IRQ signals. A value of 1 enables interrupts and a value of 0 disables them. Then we have the ipending register which indicates which of the 32 IRQ signals are currently being driven to the ISS. Each bit corresponds to one of the 32 IRQ signals and a value of 1 means there is an interrupt pending. A 0 indicates no interrupts pending.

Almost all of the instructions in the Nios II instruction set are implemented in the ISS. Those that are implemented are listed in table 4.19.

<table>
<thead>
<tr>
<th>Data transfer instructions</th>
<th>ldw, ldwio, ldb, ldbio, ldbu, ldbuio, ldh, ldhio, ldhu, ldhuio, stw, stwio, stb, stbio, sth, sthio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic and logical instructions</td>
<td>and, xor, or, nor, andi, xori, ori, andhi, orhi, xorhi, add, sub, mul, div, divu, addi, multi, mulxss, mulxu, mulxsu</td>
</tr>
<tr>
<td>Comparison instructions</td>
<td>cmpge, cmplt, cmpne, cmpgeq, cmpgeq, cmpltu, cmpgei, cmplti, cmpnei, cmpeqi, cmpeq, cmpltui</td>
</tr>
<tr>
<td>Shift and rotate instructions</td>
<td>rol, ror, roli, sl, sll, sra, srl, srai, srli</td>
</tr>
<tr>
<td>Program control instructions</td>
<td>call, callr, jmp, ret, jmpi, br, bge, blt, bne, beq, bgeu, bltu</td>
</tr>
<tr>
<td>Other control instructions</td>
<td>trap, eret, rdctl, wrctl, flushd, flushda, flushi, init, inita, initi, nextpc, sync, flushp</td>
</tr>
</tbody>
</table>

Table 4.19 Nios II instructions

The following instructions are not implemented: break, bret, rdprs, wrprs and custom instructions.

The break and bret instructions are concerned with break exceptions and are used by debuggers only. Since we don’t need a debugger here, these two instructions are not implemented. The rdprs and wrprs instructions read and write, respectively, to a specific shadow register set. Since shadow register sets are not implemented in the Nios II /s core, these two instructions are not implemented either. Finally the ISS does not support custom instructions. Custom instructions are supported in the Nios II /s core but they are never used in the IS1200 labs so they are not supported by the ISS.

The ISS supports software exceptions. A software exception is issued whenever a trap instruction is executed or an unimplemented instruction is discovered. In addition to software
exceptions there are also interrupt exceptions. An interrupt exception is an exception that is caused by a hardware interrupt. A hardware interrupt is issued when the following three conditions are met:

- Bit 0 in the status control register is one
- At least one bit in the ienable control register is one
- The same bit/bits in the ipending control register is/are one

When a software exception or hardware interrupt is issued, the following steps are taken by the ISS:

1. The status control register is copied to the estatus control register.
2. Bit 0 in the status control register is set to zero. This disables further hardware interrupts.
3. The address pointing at the next instruction the program should have executed when the exception occurred is written to register r29.
4. Execution is transferred to the exception handler.

The ISS also supports issuing a reset exception. The following actions are taken when a reset exception is issued:

1. All general purpose and control registers are set to 0.
2. The program counter is assigned the value of the reset address.

The execution cycle of the ISS can be listed as following:

1. Check for hardware interrupts. If a hardware interrupt is pending, generate a software exception which will transfer execution to the exception handler.
2. Fetch the instruction located at the address specified by the program counter.
3. Increment the program counter by 4.
4. Decode the instruction.
5. Execute the instruction.

These five steps are implemented as a function in C++ and if that function is executed over and over again it will mimic the behaviour of the Nios II/s core.

If the value of the pc points to an address that is not mapped to any device, a simulation error will occur. The user will be notified by this and the simulation will stop. The same thing will happen if a load or store instruction is executed and the address supplied is not mapped to any device or the address is not 4-byte word aligned (bit 0 and 1 of the address not being zero).

### 4.4 The I/O board

The I/O board in NIISim has been designed so that it supports not only the DE2 board, but the DE0 and DE1 boards as well. In fact, it supports any possible I/O board that has LEDs, seven segment displays, toggle switches, push buttons and an LCD display. Note that only one LCD display is supported. The user is able to add as many of the other devices as possible.

The I/O board specification is stored in a .board file (explained in section 4.1.2) which is automatically loaded when loading a system description file. The format of the I/O board
specification has been made quite general, otherwise it wouldn’t be possible to support several boards. To make the .board file format general, I developed the concept of device groups.

### 4.4.1 Device groups

The device groups are what make up the board. Without device groups you can’t add any board devices (LEDs, buttons, etc). A device group is simply a group of devices. For example, on the DE2 board in figure 3.3, all the toggle switches belong to the same device group, all the red LEDs belong to the same device group, etc. The idea behind device groups is to group the board devices together so that they can be mapped to a PIO interface. Mapping a device group to a PIO interface let’s the simulated program control the devices in that device group. Table 4.20 lists all device groups on the DE2 board in figure 3.3.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type (I = in, O = out)</th>
</tr>
</thead>
<tbody>
<tr>
<td>REDLEDs</td>
<td>O</td>
</tr>
<tr>
<td>GREENLEDs</td>
<td>O</td>
</tr>
<tr>
<td>HEXLOW</td>
<td>O</td>
</tr>
<tr>
<td>HEXHIGH</td>
<td>O</td>
</tr>
<tr>
<td>PUSH</td>
<td>I</td>
</tr>
<tr>
<td>TOGGLE</td>
<td>I</td>
</tr>
</tbody>
</table>

Table 4.20 DE2 board device groups

As can be seen, each device group has two properties. A name and a type. The name is an identifier used when adding board devices to the device group, and when mapping the device group to a PIO interface. The type tells NIISim what kind of PIO interface the device group can be mapped to. The type of the device group and the PIO interface must match. For example, the PIO interface to the red LEDs is of type “out” because it is sending data out to the LEDs. Therefore, the device group that contains the red LEDs must also be of type “out”.

Internally, a device group also has a 32-bit register, just like the PIO interfaces. When mapped, the bits in the PIO interface register correspond to the same bits in the device group register.

### 4.4.2 Board devices

Board devices are the devices you actually see on the real DE2 board. That is the LEDs, buttons, etc. When adding a board device, one must specify the type of the board device, the device group it should be added to, and what bit in the 32-bit device group register it should be mapped to. Other parameters that must be specified are a picture and coordinates. The picture is used to visualize the device on the board and the coordinates are needed to position the picture. This way, the user has complete control over the design of the board. He can choose whatever picture he wants for the buttons and where to position them.

Table 4.21 lists all valid identifiers that specify the type of the board device.

<table>
<thead>
<tr>
<th>Identifier (type)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LED</td>
<td>A normal LED.</td>
</tr>
<tr>
<td>SSLED</td>
<td>A seven segment display.</td>
</tr>
<tr>
<td>PUSH</td>
<td>A push button.</td>
</tr>
<tr>
<td>TOGGLE</td>
<td>A toggle switch</td>
</tr>
</tbody>
</table>

Table 4.21 Board device identifiers
The different devices are explained in a bit more detail below.

**LED**

The LED type is a light emitting diode. It can be either on or off. Thus, it has two different states and one bit is enough to store this state. A value of 0 is off and a value of 1 is on. A total of 32 LEDs can be added to one device group since the register has 32 bits. Figure 4.4 shows an example picture of an LED.

![Figure 4.4 An LED](image)

The picture can be of any size. The only requirement is that its width is an even number so that it is divisible by two. This is because the left part of the picture represents its off state and the right part represents its on state. When NIISim draws the picture, the corresponding bit in the device group register is used as an index to select which part to show. 0 (off) means the left part and 1 (on) means the right part.

**PUSH**

The PUSH type is a push button. A push button is a button that goes back to its default state when released. The button can be either pressed or released. Thus, it has two different states and one bit is enough to store this state. A value of 1 means released and a value of 0 means pressed. The reason why 1 means released is due to the way the button is physically connected on the read board. A total of 32 push buttons can be added to one device group since the register has 32 bits. Figure 4.5 shows an example picture of a push button.

![Figure 4.5 A push button](image)

The picture can be of any size. The only requirement is that its width is an even number so that it is divisible by two. This is because the left part of the picture represents its pressed state and the right part represents its released state. When NIISim draws the picture, the corresponding bit in the device group register is used as an index to select which part to show. 0 (pressed) means the left part and 1 (released) means the right part.

**TOGGLE**

The TOGGLE type is a toggle switch. A toggle switch is a switch that can be in two different states, up or down. Thus, one bit is enough to store this state. A value of 1 means up and a value of 0 means down. A total of 32 toggle switches can be added to one device group since the register has 32 bits. Figure 4.6 shows an example picture of a toggle switch.

![Figure 4.6 A toggle switch](image)

The picture can be of any size. The only requirement is that its width is an even number so that it is divisible by two. This is because the left part of the picture represents the down state
and the right part represents the down state. When NIISim draws the picture, the corresponding bit in the device group register is used as an index to select which part to show. 0 (down) means the left part and 1 (up) means the right part.

**SSLED**

The SSLED type is a seven segment display. It stands out of the other 3 devices because it contains 7 LEDs, thus is has 128 different states. 7 bits are needed to store the state. The bit specified when adding a seven segment display is least significant bit in the 7 bit state register. The remaining 6 bits are then mapped in a sequential order to the device group register. For example, if one maps an SSLED to bit 7 in the device group register, bit 0 in the state register is mapped to bit 7 in the device group register, bit 1 in the state register is mapped to bit 8 in the device group register, and so on. A total of 4 seven segment displays can be added to one device group. Figure 4.7 shows an example picture of a seven segment display.

![Figure 4.7 A seven segment display](image-url)
At a first glance, it looks quite complicated. This is because it has all the 128 different states in it. The picture can be of any size. The only requirement is that the width is a multiple of 8 and the height is a multiple of 16. We see that the picture has 8 columns and 16 rows. For NIISim to be able to pick out which part to draw, the 7 bits in the state register are split up in two parts, one to specify the column and one to specify the row. Figure 4.8 illustrates this.

![State register parts](image)

We see that bytes 0-2 of the state register represent the column part and bytes 3-7 represent the row part. In the picture, the value of the row part is 15 and the value of the column part is 1. If we look at row 15 and column 1 (counting starts at 0, not 1) we see the number 1 displayed. Hence, if one were to write the value 0x79 to a seven segment’s state register, NIISim would display the part of the picture representing the digit 1.

### 4.4.3 The LCD display

The last device that can be added to an I/O board is an LCD display. It is different from the other board devices because it does not require a device group. This is mainly because it is not mapped to a PIO interface. It is mapped to an LCD interface. NIISim only supports one LCD display. When adding an LCD display one has to specify a name identifier and coordinates. The identifier is used when mapping the LCD display to an LCD interface and the coordinates are used to position the device on the board. No picture is used to visualize the device. Instead, a standard windows edit box is created which supports two rows each having space for 16 characters, allowing for a total of 32 characters.

### 4.5 Threads

This section will discuss the two threads that run in parallel with the NIISim main program. The main program here is essentially the windows message loop. The windows message loop is what handles all the messages so that mouse clicks, button presses and all other window specific actions work. In parallel with this message loop are two threads that perform the actual simulation. The first thread, called the simulation thread, is the one that performs the actual simulation. The second thread, called the update thread, is the one that updates the text in all console windows. It also updates the register window and the text in the LCD display on the I/O board.

#### 4.5.1 Simulation thread

To achieve high speed simulation, the simulation cycle has been placed in a separate thread. The simulation thread consists of an infinite loop. Each cycle in this loop checks if the simulation is running. If it is running, an instruction is executed. Otherwise it pauses for 10 milliseconds so that the thread does not use up all of the processing power of the CPU that is running NIISim. Then the check if the simulation is running is performed again and so on. This is illustrated in Figure 4.9.
It may seem a bit strange that we are waiting for 0 ms when not executing in slow mode. However, in Windows this is necessary because if we didn’t perform the wait, the simulation thread would block both the Windows message loop and the update thread. The wait for 0 ms statement simply tells the processor that the simulation thread is done for now so let the other threads execute. If we are executing in slow mode, we perform a wait of 2 ms. This will lock the simulation to 500 instructions per second.

The simulation thread is spawned at the same time as NIISim is started and it will stay active until the user exists NIISim. While the simulation is stopped or paused, the simulation thread will be stuck in the upper loop, checking for the simulation state and performing a wait of 10 ms if necessary. This wait will cause the thread to be inactive most of the time when the simulation isn’t running. We don’t want the thread to use up CPU processing power when there is nothing to simulate.
4.5.2 Update thread

The update thread performs the updating of the three console windows, the register window and the I/O board LCD display. When the simulated software writes something to a console or the LCD display, they are not updated immediately. If they were, it would slow down the simulation quite a lot. There is no need to update the consoles after each character has been written to them. Instead, the characters are placed in a buffer. The update thread checks this buffer for new data and prints it out. Figure 4.10 illustrates this.

![Update thread flowchart](image)

As can be seen, the consoles are only updated if their respective text buffers are not empty. Also note that the register window is only updated if it is actually visible. The wait of 100 ms causes this loop to be run 10 times each second. This is enough to not use up too much unnecessary CPU processing power but still update the consoles at a reasonable rate.

This thread, just like the simulation thread, is spawned when NIISim starts and remains active until the user exits NIISim.

4.6 Trace file generation

NIISim has a feature that can generate a trace file containing all memory accesses that have occurred. The Dinero [13] trace file format has been chosen as the format for the trace file. Dinero is a trace-driven cache simulator. Trace-driven means it needs a memory trace file as input as opposed to NIISim which is execution-driven and requires the machine code of the software as input.
When trace file generation has started, NIISim will store all memory reads, writes and instruction fetches in the created .din file. It is very important for the user to be aware that the size of the trace file might grow very fast. Thus, the trace file generation feature should not be used for long periods of time when simulating in fast mode. It is better to use it when running in slow mode.

### 4.6.1 Dinero file format

The Dinero trace file (.din file) is a plain text file, similar to the .sdf file. Each line in the .din file corresponds to a 32-bit memory transfer. The line is divided into two parts separated by a space. The first part is a one digit number that tells Dinero if the memory transfer is a read, write or instruction fetch. The numbers are listed below.

- 0 – Memory read
- 1 – Memory write
- 2 – Instruction fetch

The second part is the address. It is a 4 byte hexadecimal number without the prefix 0x. The listing below is an excerpt from a .din file generated by NIISim.

```
2 008001F0
2 008001F4
2 008001EC
1 0080F458
2 008001F0
2 008001F4
2 008001EC
1 0080F45C
2 008001F0
2 008001F4
2 008001EC
1 0080F460
```

It is safe to assume that this is a loop that is executing. The instructions at addresses 008001F0, 008001F4 and 008001EC keep repeating. In between them there are memory writes at increasing addresses.

### 4.7 Command line parameters

NIISim has two command line parameters that allow the user to automatically load an .sdf file and an .elf file when NIISim is started. They are explained in table 4.22.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-s</td>
<td>This parameter loads an .sdf file. The name of the .sdf file to load is specified after the parameter. There must be a space in between the parameter and the filename. The filename can be either a full path or a path relative to the NIISim executable program.</td>
</tr>
<tr>
<td>-e</td>
<td>This parameter loads an .elf file. The name of the .elf file to load is specified after the parameter. There must be a space in between the parameter and the filename. The filename can be either a full path or a path relative to the NIISim executable program.</td>
</tr>
</tbody>
</table>

Table 4.22 Command line parameters
Note that it is not possible to load an .elf file without first loading an .sdf file. If the user wants to load both files, the -s parameter must come before the -e parameter. An example would be:

NIISim.exe -s "is1200 system.sdf" -e testboard.elf

This will load an .sdf file called “is1200 system.sdf” and an .elf file called “testboard.elf”, both located in the same directory as the NIISim executable program. Since the name of the .sdf file contains a space character, double quotes have to be used.
5. Testing

In this chapter, we will see if the IS1200 course labs do work in NIISim. A comparison with the simulator from Altera, Nios II ISS, will also be done. Is NIISim faster or slower than Nios II ISS?

5.1 The IS1200 labs

In the IS1200 course, there are four labs with utilize the Altera DE2 Board. The first lab is called nios2time and mainly focuses on assembly programming. The second one is called nios2io. It focuses on using the I/O devices on the board. The third lab is called nios2int and focuses on interrupts. The fourth and final lab is a lab that does not require the DE2 Board but still uses the Nios II ISS. It focuses on threads and synchronization.

Each lab is divided into a couple of smaller assignments. All those assignments have been tested and I have found that they can all be performed NIISim. One assignment is the last assignment in the nios2int lab. The results of that assignment are shown in figure 5.1.

![Figure 5.1 nios2int lab results](image)

In this lab assignment, the software is constantly printing out underscores ‘_’ in the UART0 console. A hardware timer is updating a counter variable using interrupts. Every time the counter counts up by 1, its value is printed out in the UART0 console in the format XX:XX and on the four rightmost seven segment displays. No line breaks are printed which is why everything appears on one line.
Pressing the rightmost switch causes the character ‘S’ to be printed out, and after a short delay, the character ‘s’ is also printed out. This can be seen in figure 5.1. The rightmost red LED also lights up when this happens. Similarly, when pressing the rightmost push button, the character ‘D’ is printed out. After a short delay, the character ‘d’ is printed out. The rightmost green LED lights up when this happens. All the printing is done using interrupts and the push button has a lower IRQ number meaning higher priority.

The key point in this assignment is nested interrupts. After pressing the switch and the delay has started, the user is supposed to press the push button so that it interrupts the delay function. We can confirm by looking at figure 5.1 that this has indeed happened. The character ‘D’ has been printed out meaning the software is then executing the delay loop. Then we see the character ‘S’. This means the push button has been pressed during the execution of the first delay. Hence, nested interrupts are working correctly. We also see both the red and green LEDs on as they should be.

Output from the fourth lab about threads and synchronization can be seen in figure 5.2.

![UART0 Console](image)

Figure 5.2 Results from The threads and Synchronization lab

The software here spawns threads that produce prime numbers. Thread switching is performed and everything is printed to the UART0 console instead of the JTAG UART console.

5.2 Comparison with the Nios II ISS

Here we will try to measure how fast NIISim is compared to the Nios II ISS. To do this, a test program has to be executed on both simulators. The test will be performed in both Windows and Linux. A flowchart of the test program can be seen in figure 5.3. The full source code can be found in appendix A.
The test program is an infinite loop. The loop consists of printing out the value of a counter and then waiting for a timer to timeout. The period of this timer will be set to 50000000 which corresponds to the 50 MHz clock on the real DE2 board. If one were to execute this program on the real board, the delay between each printout of the counter would be exactly 1 second. When running the program on the simulators and measuring the time between each printout, we get an idea of how well they perform compared to each other and the real board.

The results of the tests are shown in table 5.1.

<table>
<thead>
<tr>
<th>Windows</th>
<th>Time</th>
<th>Linux (Ubuntu)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIISim</td>
<td>18 seconds</td>
<td>NIISim</td>
<td>30 seconds</td>
</tr>
<tr>
<td>Nios II ISS</td>
<td>124 seconds</td>
<td>Nios II ISS</td>
<td>95 seconds</td>
</tr>
</tbody>
</table>

Table 5.1 Test results

We see here that NIISim is almost 7 times faster than Nios II ISS. This is quite remarkable. A question that immediately arises is how come Nios II ISS simulates software so slow compared to NIISim when developing time has not been focused on optimizing NIISim for speed? One answer to that question is because Nios II ISS is originally written for Linux, not Windows. When Nios II ISS is run on Windows, it uses Cygwin dlls to simulate the Linux calls. This, of course, causes the simulation to be slower in Windows than in Linux.

When looking at the results for the Linux tests it should be noted that Ubuntu was run in a virtual machine, not on actual hardware. Due to this, the Windows and Linux results cannot be directly compared. The Linux results should be a bit better if run on real hardware. However, it is still notable that the Nios II ISS executes the test program faster in Linux than on Windows. This confirms the fact that Nios II ISS is indeed written for Linux and not for
Windows. It is also interesting to see that even though Nios II ISS is written for Linux, NIISim is still faster even though it has to be run using Wine (a program used to run Windows executables under Linux).

These results have one important aspect. In some of the IS1200 labs there are timers that are initialized with periods that matches 100ms when run on the real board. When simulating these programs, one has to wait very long before one sees any output. This is because 100 ms takes roughly 124s / 10 = 12.4s to simulate. To speed this up, the student often has to keep changing the period of the timer when simulating the program on the PC and then change it back when running it on the board. This can be quite tedious in the long run.

Using NIISim, simulating 100 ms only takes around 18s / 10 = 1.8s which is much more acceptable than 12.4s. The student probably won’t have to change the period of the timer if he uses NIISim to perform the simulation. That saves a lot of time as he won’t have to keep recompiling the program.
6. Conclusions
In this chapter we will look back on the project objectives and requirements. Did all the objectives get completed? Does NIISim satisfy all the requirements that were set up? These questions are important. If an objective was not completed then it is important to ask yourself why, and if something could have been done in another way so that the objective did get completed. Or could it be that the project objectives were set way too high? Some future work on NIISim will also be discussed.

6.1 The project objectives and requirements
The main objectives in this thesis were:

- Develop a simulator platform that can simulate the Altera DE0, DE1 and DE2 boards and CPU caches.
- Make sure all labs, apart from the ones that involve the serial port, in the IS1200 course run without bugs in the simulator.
- Make sure that the cache simulator can be used to complete the cache lab in IS1200.

Looking at the first objective we can conclude that part of it has been completed. The Altera DE0, DE1 and DE2 board can all be simulated. However, no cache simulator was implemented. It is not possible to simulate CPU caches. The reason why the cache simulator fell behind was due to lack of time. Since NIISim had to be developed completely from scratch, a huge part of the developing time had to be spent on that, so there was simply not enough time left for the cache simulator.

The second objective has definitely been completed. All the IS1200 labs have been tested and works fine in NIISim. It is also possible to simulate serial ports (using UART interfaces) even though I initially thought that would be very difficult.

There is not much to say regarding the third objective since it has already been stated that the cache simulator part of NIISim has not been implemented at all. Hence, this objective has not been completed.

There was also a fourth optional objective which said that if all 3 objectives above were completed, a pipeline simulator should be implemented. Since it is already concluded that not all objectives did get completed, we don’t even have to discuss the forth one.

If we look at the project requirements we see that many of them concern the cache simulator. We can ignore them since the cache simulator was not implemented at all. All other requirements are actually met. NIISim has been written in C++ and compiled for Microsoft Windows. It can run without problems in Linux using Wine. A pipeline simulator should not be that difficult to implement since the function that executes instructions has been separated in one instruction fetch phase, one decode phase and one execution phase. Extending NIISim to support multi-core simulation is also quite straight forward. The biggest issue will involve handling memory consistencies that will occur if several CPUs write to the same memory location in the same clock cycle. And finally, the I/O devices that can be simulated are only those that are preset in the IS1200 labs.

Overall I think the thesis is a success. A fully functional product has been developed. It is reliable and it can be used to simulate Altera systems and I/O boards based on a Nios II CPU.
6.2 Future work

There are many ways one could improve NIISim. The first thing I think of is adding a cache simulator. Since the cache simulator did not get implemented, it is a perfect thesis project for another student to work on.

6.2.1 Cache simulator

Implementing a cache simulator will require a lot of work. Not only will one have to figure out how to store the cache data at the programming level, but one also have to design a good way of visualizing the cache data. Visualizing cache data was touched upon in section 2.3. Deciding on how many of the cache parameters the user should be able to change is also important. This was discussed in the introduction chapter. Some requirements were even set up concerning the cache parameters. If one is to implement a cache simulator into NIISim, studying chapter 1 and 2 in this report is probably a good idea. It will give that person some ideas and hopefully he will improve them.

Adding a cache simulator to NIISim could be done by adding a new button to the toolbar. This was the initial idea.

6.2.2 More debugging windows

There are many sorts of child-windows that could be added to NIISim. Two of them were discussed during the development of NIISim but they didn’t get implemented. Those two were a disassembly window and a memory window. In the disassembly window one would be able to see the assembler instructions of the software. One would be able to follow the execution very easily. Symbols could be added to help the user know what functions that are currently being simulated. Symbol tables exist in the .elf files.

Breakpoints go hand in hand with a disassembly listing and they would make the trace file generation much more useful. For example, one would be able to set breakpoints at the start and at the end of a function that one wants a memory trace of. When the breakpoints are set, the simulation will start and then stop at the beginning of the function. At this point, the user turns on trace file generation and starts the simulation again. The simulation stops at the second breakpoint and at that point the user stops trace file generation. The trace file will then contain memory traces that occurred only within that function.

Another window that could be useful is a memory window. Using the memory window one would be able to look directly at the data in the SDRAM memories. It could be expanded to cover the entire address space. When viewing the memory contents like this, an option that allows the user to change the data in real time could also be added.

6.2.3 Multi-core simulation

NIISim has been designed so that it should be quite straight forward to add support for multi-core simulation. It was one of the requirements. It is already possible to add multiple CPUs to the system. However, only the first CPU that was added is the one that executes the simulated program. Extending it to support multi-core simulation would simply involve creating a loop that would go through all CPUs in the system and letting each of them execute one instruction. This loop would be placed in the simulation thread, replacing the code that currently only executes one instruction from the first CPU.
A problem that arises with this implementation is memory consistency. For example, let’s say we have a system with two CPUs, and in the same clock cycle CPU1 writes data to memory location X and CPU reads data from the same memory location X. If the CPUs are simulated in a sequential order where CPU1 executes an instruction before CPU2, CPU2 will read the value CPU1 wrote to location X. This is something that might not happen in real hardware. There are also other situations where both CPUs would write data to the same memory location, all in the same clock cycle. Which data would then be written to that memory location? These will be the most important problems to solve when working on multi-core simulation.
7. References


### Appendix A. Test program

```c
#include <stdio.h>
#include "system.h"

#define TIMER1 ((volatile int *) TIMER_1_BASE )

void set_period(int period)
{  
    volatile int * p = TIMER1;
    // Set the period
    p[2] = period & 0xFFFF;
    p[3] = (period & 0xFFFF0000) >> 16;
}

int main()
{
    volatile int * p = TIMER1;
    int counter = 0;

    // Set period to 50000000 (corresponding to 50MHz)
    set_period(50000000);

    // Wait for user to press a key
    printf("Press a key to start\n");
    getchar();

    // Start timer
    p[1] = 7;

    // Main loop
    while(1)
    {
        // Print number of seconds passed
        printf("%d seconds\n", counter);

        // Count up
        counter++;

        // Wait for timer to timeout
        while(!(p[0] & 1)) {};

        // Clear the timeout bit
        p[0] = 0;
        // Start the timer again
        p[1] = 7;
    }

    return 0;
}
```
Appendix B. Example .sdf file

AddCPU "cpu", 0x8000000, 0x8000200, 50000000

AddSDRAM "sdram", 0x8000000, 0x8000000

AddUART "uart_0", 0x860, 0x20, 4
AddUART "uart_1", 0x880, 0x20, 5
AddUART "uart_2", 0x8A0, 0x20, 17

AddJTAG "jtag_uart", 0x800, 0x8

AddLCD "character_lcd", 0x808, 0x8

AddTimer "timer_0", 0x820, 0x20, 50000000, 1, "ms", 0, 0, 1, 1
AddTimer "timer_1", 0x840, 0x20, 50000000, 1, "ms", 0, 0, 1, 10
AddTimer "timer_2", 0x860, 0x20, 50000000, 1, "ms", 0, 0, 1, 11
AddTimer "timer_3", 0x880, 0x20, 50000000, 1, "ms", 0, 0, 1, 12
AddTimer "timer_4", 0x8A0, 0x20, 50000000, 1, "ms", 0, 0, 1, 13
AddTimer "timer_5", 0x8C0, 0x20, 50000000, 1, "ms", 0, 0, 1, 14
AddTimer "timer_6", 0x8E0, 0x20, 50000000, 1, "ms", 0, 0, 1, 15

AddPIO "redled18", 0x810, 0x10, "out"
AddPIO "keys4", 0x840, 0x10, "in", 2
AddPIO "toggles18", 0x850, 0x10, "in", 3
AddPIO "hex_low28", 0x860, 0x10, "out"
AddPIO "hex_high28", 0x900, 0x10, "out"
AddPIO "greenled9", 0x910, 0x10, "out"

AddPIO "jp1_in1_5", 0x8A0, 0x10, "in", 6
AddPIO "jp1_in2_8", 0x8B0, 0x10, "in", 7
AddPIO "jp1_in3_5", 0x8C0, 0x10, "in", 8
AddPIO "jp1_out1_5", 0x8D0, 0x10, "out"
AddPIO "jp1_out2_8", 0x8E0, 0x10, "out"
AddPIO "jp1_out3_5", 0x8F0, 0x10, "out"

ImportBoard "boards\de2.board"

Map "jtag_uart", "JTAG"
Map "uart_0", "UART0"
Map "uart_1", "UART1"
Map "redled18", "REDLEDS"
Map "keys4", "PUSHBUTTONS"
Map "toggles18", "TOGGLESWITCHES"
Map "hex_low28", "HEXLOW"
Map "hex_high28", "HEXHIGH"
Map "greenled9", "GREENLEDS"
Map "character_lcd", "LCD"
Appendix C. Example .board file

SetName "DE2 Board"
SetBackgroundImage "images\bg.bmp"

AddDeviceGroup "REDLEDS", "out"
AddDevice "LED", "REDLEDS", 0, "images\redled.bmp", 355, 236
AddDevice "LED", "REDLEDS", 1, "images\redled.bmp", 335, 236
AddDevice "LED", "REDLEDS", 2, "images\redled.bmp", 315, 236
AddDevice "LED", "REDLEDS", 3, "images\redled.bmp", 295, 236
AddDevice "LED", "REDLEDS", 4, "images\redled.bmp", 275, 236
AddDevice "LED", "REDLEDS", 5, "images\redled.bmp", 255, 236
AddDevice "LED", "REDLEDS", 6, "images\redled.bmp", 235, 236
AddDevice "LED", "REDLEDS", 7, "images\redled.bmp", 215, 236
AddDevice "LED", "REDLEDS", 8, "images\redled.bmp", 195, 236
AddDevice "LED", "REDLEDS", 9, "images\redled.bmp", 175, 236
AddDevice "LED", "REDLEDS", 10, "images\redled.bmp", 155, 236
AddDevice "LED", "REDLEDS", 11, "images\redled.bmp", 135, 236
AddDevice "LED", "REDLEDS", 12, "images\redled.bmp", 115, 236
AddDevice "LED", "REDLEDS", 13, "images\redled.bmp", 95, 236
AddDevice "LED", "REDLEDS", 14, "images\redled.bmp", 75, 236
AddDevice "LED", "REDLEDS", 15, "images\redled.bmp", 55, 236
AddDevice "LED", "REDLEDS", 16, "images\redled.bmp", 35, 236
AddDevice "LED", "REDLEDS", 17, "images\redled.bmp", 15, 236

AddDeviceGroup "GREENLEDS", "out"
AddDevice "LED", "GREENLEDS", 0, "images\greenled.bmp", 474, 152
AddDevice "LED", "GREENLEDS", 1, "images\greenled.bmp", 454, 152
AddDevice "LED", "GREENLEDS", 2, "images\greenled.bmp", 434, 152
AddDevice "LED", "GREENLEDS", 3, "images\greenled.bmp", 414, 152
AddDevice "LED", "GREENLEDS", 4, "images\greenled.bmp", 394, 152
AddDevice "LED", "GREENLEDS", 5, "images\greenled.bmp", 374, 152
AddDevice "LED", "GREENLEDS", 6, "images\greenled.bmp", 354, 152
AddDevice "LED", "GREENLEDS", 7, "images\greenled.bmp", 334, 152
AddDevice "LED", "GREENLEDS", 8, "images\greenled.bmp", 314, 152

AddDeviceGroup "HEXLOW", "out"
AddDevice "SSLED", "HEXLOW", 0, "images\7segled.bmp", 273, 163
AddDevice "SSLED", "HEXLOW", 7, "images\7segled.bmp", 242, 163
AddDevice "SSLED", "HEXLOW", 14, "images\7segled.bmp", 211, 163
AddDevice "SSLED", "HEXLOW", 21, "images\7segled.bmp", 180, 163

AddDeviceGroup "HEXHIGH", "out"
AddDevice "SSLED", "HEXHIGH", 0, "images\7segled.bmp", 114, 163
AddDevice "SSLED", "HEXHIGH", 7, "images\7segled.bmp", 83, 163
AddDevice "SSLED", "HEXHIGH", 14, "images\7segled.bmp", 44, 163
AddDevice "SSLED", "HEXHIGH", 21, "images\7segled.bmp", 13, 163

AddDeviceGroup "PUSHBUTTONS", "in"
AddDevice "PUSH", "PUSHBUTTONS", 0, "images\pushbutton.bmp", 454, 177
AddDevice "PUSH", "PUSHBUTTONS", 1, "images\pushbutton.bmp", 414, 177
AddDevice "PUSH", "PUSHBUTTONS", 2, "images\pushbutton.bmp", 374, 177
AddDevice "PUSH", "PUSHBUTTONS", 3, "images\pushbutton.bmp", 334, 177

AddDeviceGroup "TOGGLESWITCHES", "in"
AddDevice "TOGGLE", "TOGGLESWITCHES", 13, "images\toggleswitch.bmp", 92, 261
AddDevice "TOGGLE", "TOGGLESWITCHES", 14, "images\toggleswitch.bmp", 72, 261
AddDevice "TOGGLE", "TOGGLESWITCHES", 15, "images\toggleswitch.bmp", 52, 261
AddDevice "TOGGLE", "TOGGLESWITCHES", 16, "images\toggleswitch.bmp", 32, 261
AddDevice "TOGGLE", "TOGGLESWITCHES", 17, "images\toggleswitch.bmp", 12, 261
AddLCD "LCD", 15, 65