Laboratory Exercises for EITF20 Computer Architecture

Anders Ardö, Mohammad Attari
Department of Electrical and Information Technology
Lund University

November 10, 2019

Contents

LUND INSTITUTE OF TECHNOLOGY
Lund University

In the course of these labs you will run a large number of simulations, and it may be difficult to keep track of your results unless you maintain a lab book (hard copy or on-line). This book should contain the description of all the simulation runs you performed, your simulation plans, comparison of results, graphs if any etc. In addition as you will start using more detailed simulators, the simulation time will increase. A lab-book, which documents all the simulation runs you performed already, will help you avoid repeat runs and will save you considerable time. The system allows you to attach an arbitrary ID to each run, as well as it saves all results within a session (typically one lab). Use this intelligently in order to keep better track of what you have done.

The software tools used in this laboratory are of an educational nature. In plain English this means that one may expect all kind of problems with the tools themselves, the installation and the assistants, because 'things can have changed since last time'. We will give you no other guarantee than all the help we can.

Written solutions to home assignments for a lab should handed in to the lab-assistant before the lab starts.

1 Laboratory 1: Pipelined Processors

NOTE! All software tools are available at `http://dark.eit.lth.se/`.
If prompted, use your credentials as follows:

username: Your Lucat-ID
password: Your Password

In this lab you need to use a Windows 7 virtual machine in order to run the MipsIt toolchain. The virtual machine for Windows 7 is located in the **C:\Temp** folder of your local computer (the file `MipsIt2000.vbox`). If by any chance the VM files are not there, you can copy the needed files (two files) from **T:\Unprotected\Masoud** to the local **C:\Temp**. Fire up the virtual machine (by double clicking on `MipsIt2000.vbox` and starting it in the VM manager), and log into the Windows 7 running on the VM with the following:

username: User
password: P@ssw0rd

As the virtual machines are bound to the local computer, we recommend every student to store their project files in a directory on their home drive! In order to do this, on the desktop you will find a shortcut to the **Map Drives** utility (that maps your student home drive to the OS running on the VM). Please run Map Drives, and provide your Lucat credentials. Check to make sure you have access to your home drive (H:), and use this drive to create your projects in, so that you can have access to them regardless of what machine you are working on.

1.1 Goals

In this laboratory exercise we practice the pipelining techniques introduced during the lectures. We start things off with the classical pipeline to see how stalls can create the so-called bubbles inside the pipeline. We then introduce forwarding to mitigate the effect of stalls on program exectution. The next step is to move on to the more advanced *ILP* techniques, such as *score-boarding* and the *Tomasulo algorithm*. To accomplish these, we will take advantage of a number of different educational tools and software especially developed to illustrate the above-mentioned concepts.

After finishing this laboratory exercise, you should be able to explain the basic principles of pipelining, including the concepts of data and control hazards, and possible remedies for them, such as forwarding and delayed branches. Finally, you should form a solid understanding for how the instructions are used to control different parts of the data path through the control unit.

1.2 Literature

Hennessy and Patterson, Computer Architecture: A Quantitative Approach, 6th edition, Appendix C, Chapter 3
MIPS Lab Environment Reference Manual (Section A.1 in this lab-manual)
To get an in-depth understanding of the pipeline you can refer to:
Hennessy and Patterson, Computer Organization and Design MIPS Edition: The Hardware/Software Interface, Chapter 4

1.3 Preparations

Carefully read the literature and all laboratory exercises below, and perform the home assignments. Note that you must solve the home assignments, or you will not be allowed to start the laboratory exercise.

1.3.1 Home Assignments

NOTE! Written solutions to the home assignments must be handed in to the lab assistant before you start the lab! This applies to all of the 4 labs.

- What is a CPU (or processor)?
- What is an assembly language program?
- How does a computer execute simple machine language instructions?
- What is the relation between assembly language and machine language?
- What does the instruction '`add t0, t1, t2`' do?
- What does the instruction '`beq t0, t1, Dest`' do?
- How does a pipelined CPU differ from a non-pipelined one?
- Describe hazards?

1.4 Introduction to the MipsIt Environment

The first experiments will be performed using `MipsIt` (found here: `https://www.eit.lth.se/fileadmin/eit/courses/eitf20/Labs2016/MipsICT.exe.zip`) for the analysis and linkage of programs and `mipspipe2000` for the subsequent execution in the pipeline. `MipsIt` is a Windows integrated development environment (IDE) for MIPS processor. Figure 1 shows the IDE in action.

If you have used Microsoft Developer Studio/Visual C++ IDE or similar, you should have a pretty good idea how `MipsIt` works. But if you are new to IDEs, you need to know what a project is. A project is a collection of interrelated source files that are compiled and linked to make up an executable file that, in our case, can be uploaded to the simulator. A project may also include text files for informational purposes.

1.4.1 IDE Basics

The IDE consists of the following panes (see Figure 1):

- The project view that contains a list of files included in the project. To open a file for editing simply double click on it in the list.
- The output window where all the output from building the project is printed out.

Many commands also have hot-keys (like most Windows programs) to make work more efficient. There is also a toolbar for easier access. Some commands are currently not implemented, and therefore are disabled.

To configure the IDE, go to File -> Options... . You can change COM settings, compiler executable, paths etc. When you start `MipsIt` for the first time it will normally auto-configure

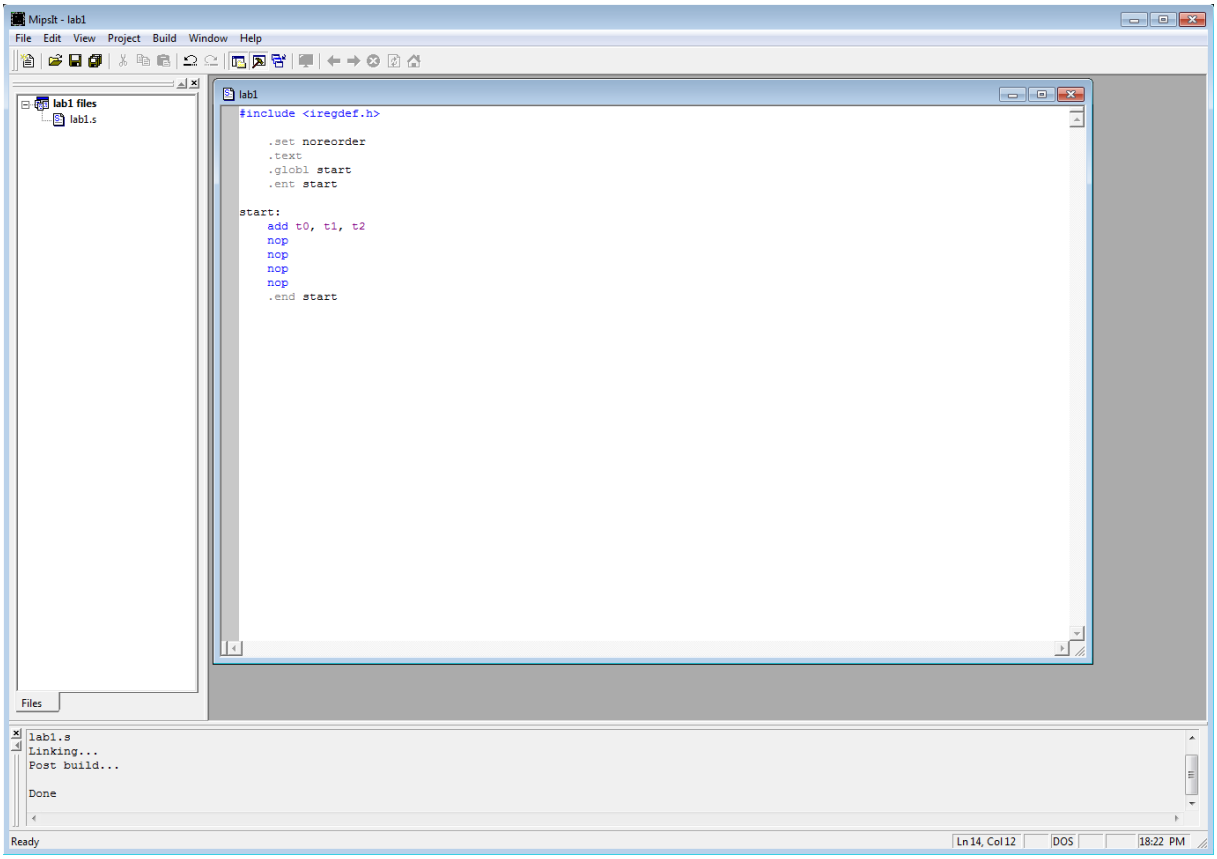

Figure 1: MipsIt.

correctly, except that it will complain about not being able to open the COM-port. *This can normally be ignored - just click OK.*

1.4.2 Creating a Project

To create a new project follow these steps:

- Choose 'New' from the 'File' menu, and then click the Project tab (if it is not already selected) in the resulting New dialog box shown in Figure 2.
- Select the type of project you want to create from the list. The project types are as follows:
  1. Assembler - If your project will only contain assembly files. **You need to select this choice for this lab!**
  2. C/Assembler - If you want a project that will contain only C or C and assembly files.
  3. C(minimal)/Assembler – Same as C/Assembler except with minimal libraries. This will be your choice (in lab 3) if you want a project that contains C files.
  4. Enter a name for the project and change the location if desired, and then click OK.

The differences between the project types are the default libraries and modules that will be included. A C/Assembler project will link with a couple of libraries, and will result in a bigger executable (which **will not work with this simulator**). A C(minimal)/Assembler

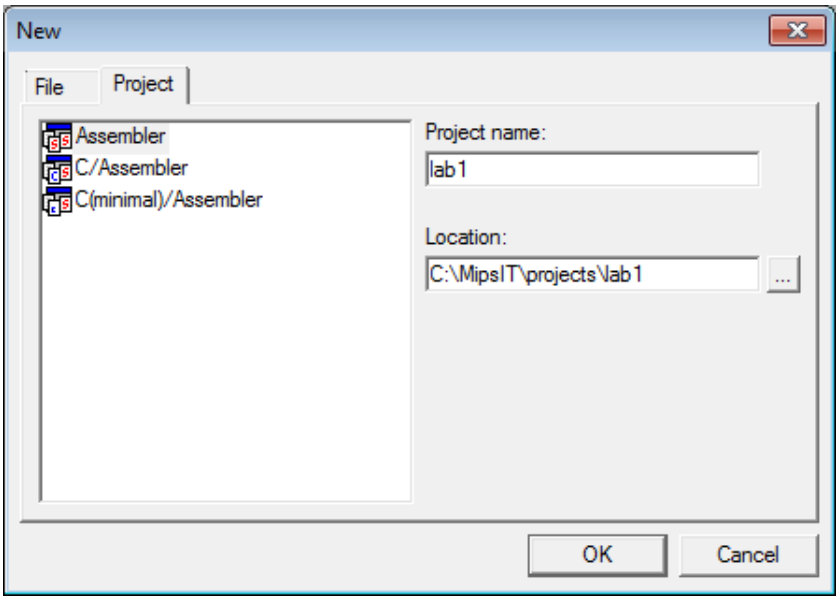

Figure 2: The 'New' dialog window (with Project tab selected).

project will link with only the bare minimum libraries, and therefore would result in a smaller executable compared to the one generated for C/Assembler projects (this one **will work with our simulator**).

1.4.3 Adding Files to a Project

If you followed the steps for creating a new project, you should now have an empty project created. You can now add files to it, by either creating new files or adding existing files. Creating a new file is similar to creating a new project, except that you select the File tab (see Figure 3) instead of the Project tab in the New dialog window. If you want to add an existing file, choose 'Add file...' from the Project menu, and then select the file you want to add. In this laboratory exercise we are only interested in assembly files (with the extension **.s**).

You can use the following assembly code as a simple example to get yourself off the ground:

```
1  #include <iregdef.h>
2
3  .set noreorder      # Avoid reordering instructions
4  .text               # Start the instructions section
5  .globl start        # The label should be globally known
6
7  .ent start          # The label marks an entry point
8  start:
9          add t0, t1, t2
10         nop
11         nop
12         nop
13         nop
14 .end start          # The label marks an exit point
```

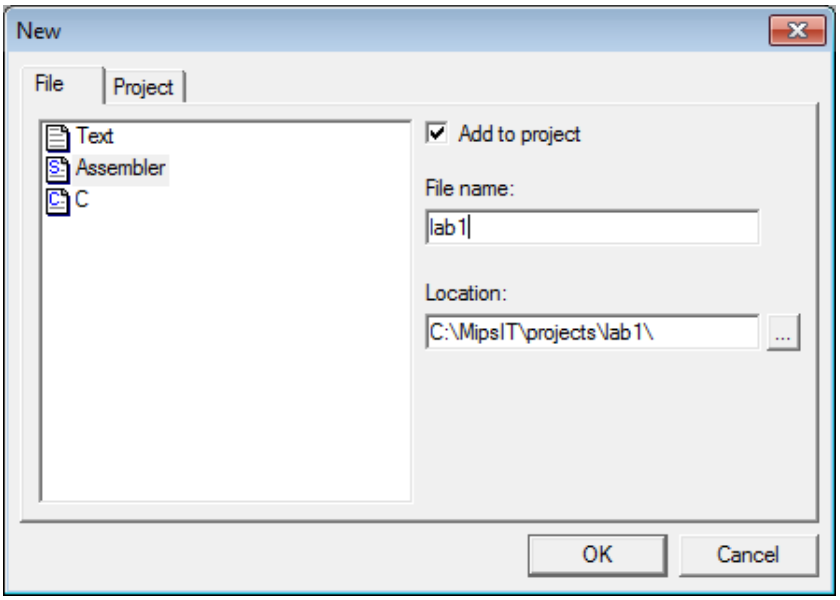

Figure 3: The New dialog window (with File tab selected).

1.4.4 Building

In order to prepare your project for simulation, choose Build from the Build menu or press F7. Any file that needs compilation will be compiled (or assembled) and finally the executable will be linked. Current status and results of the build process can be seen in the output window (see Figure 1). In case you want to re-compile all files, even those that have not been modified since the last build, choose Rebuild All from the Build menu. When the project has been successfully built, you can move to the simulator.

1.4.5 Simulation

Now open the program `mipspipe2000.exe` (in the **MipsIT\bin** folder) to run the pipe simulator. Choose 'Load Pipeline...' from the file menu, open the directory called 'S-script' and choose the file called `s.dit` to load the simple version of the pipeline. Next choose 'Open' from the file menu or click on the Open icon in the toolbar, navigate to the directory where you created your project, and open the file with the `.out` extension located in the sub-directory named 'Objects'.

Figure 4 shows the window you should be able to see if you followed the previous steps. You can play around with the tool to get a feel for it, before you move on to the next stage.

1.5 Arithmetic Instructions

Different classes of instructions use a different selection of the available components in the data path. It is common to group such instructions into four classes: arithmetic, load, store, and branch. Within one such class the instructions are quite similar, and it is often enough to understand one of them well, in order to understand them all.

The arithmetic instructions are sometimes also called register instructions, because they perform an operation with two source and one destination registers. We will now examine how an arithmetic instruction goes through the various stages of the pipeline.

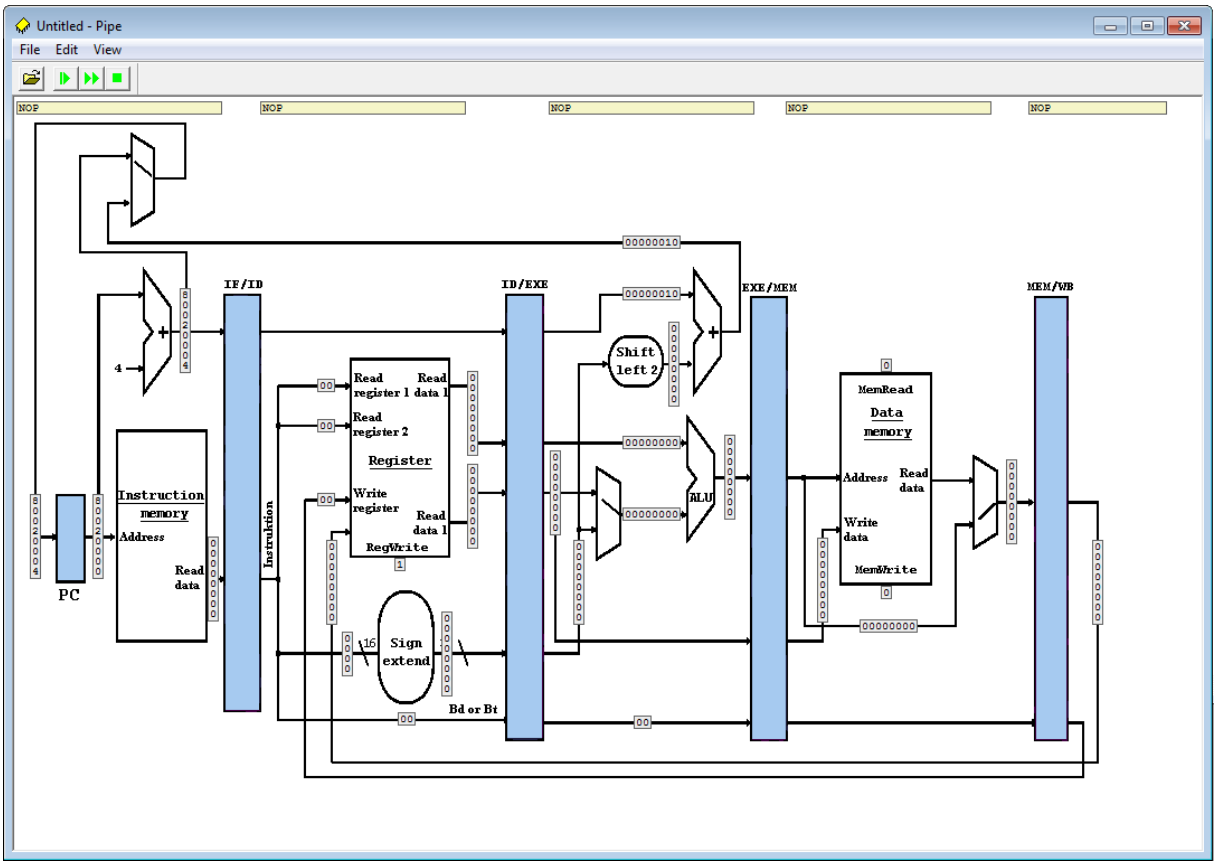

Figure 4: mipspipe2000 loaded with the simple pipeline.

1.5.1 Experiment 1

Go back to `MipsIt` and modify the assembly code to populate `t1` and `t2` with some distinct values (for example by adding `lui ...` instructions before the `add` instruction). Rebuild the project and load the program into `mipspipe2000`, and try to single-step through it, and observe what happens in each stage as the instructions flow down the pipe.

NOTE! Unfortunately you need to close `mipspipe2000` and reopen it each time you load a new program, as the tool loads the compiled program into the pipeline correctly only the first time it runs.

Now answer the following questions, while describing all signals, register changes, and other effects in detail:

- What happens when an instruction goes through the first pipeline stage, that is the IF stage?
- What happens in the second (ID) stage?
- What happens in the third (EX) stage?
- What happens in the fourth (MEM) stage?

- What happens in the fifth (WB) stage?
- What do these stage names (IF, ID, EX, MEM, and WB) stand for? What tasks do they perform?
- How many clock cycles does it take for the result of the operation to be available in the destination register? Is the result correct? How can you modify the code to make sure it is?
- In which pipeline stages do different arithmetic instructions (for instance `add` and `sub`) perform different operations?
- One stage is not used by arithmetic instructions. Which one? Why?

1.5.2 Experiment 2

Replace the `add t0, t1, t2` instruction in the program with the `lw t0, 0(t1)` instruction, build, upload, and investigate the program. Note that you must add a data variable from which to load a value.

- What happens in the different pipeline stages?
- What arithmetic operation does the ALU perform? Why?
- How many clock cycles does it take for the destination register to receive its value?
- Are all the pipeline stages used? Explain.

1.5.3 Experiment 3

Now experiment with the `sw t0, 4(t1)` instruction, and try to store a certain value that you have loaded into `t0` inside the memory with it.

- What happens in the different pipeline stages?
- What arithmetic operation does the ALU perform? Why?
- How many clock cycles does it take for the memory to receive its value?
- Are all pipeline stages used? Explain.

1.5.4 Experiment 4

Finally, investigate the use of `beq t0, t1, Dest` instruction, similar to what you did for the instructions above. Note that you must add a label named *Dest* somewhere.

- What happens in the different pipeline stages?
- What arithmetic operation does the ALU perform? Why? (There is a bug in simulator GUI. Can you spot it?)
- How many clock cycles does the instruction take to perform the jump?
- Are all pipeline stages used? Explain!

1.6 A Small Program Example

Pipelining can enable processors to potentially run up to N times faster, where N is the number of pipeline stages. However, there are several issues that make it impossible to reach this efficiency ceiling in practice. One such factor is that not all instructions will be able to use all the pipeline stages. In the following experiments we will delve a bit deeper into this.

1.6.1 Experiment 1

So far we have been looking at one instruction at a time, but the real gain with using pipelining is to be able to overlap execution of several instructions. The net effect is that the hardware is busy executing several instructions *at the same time*. We will now investigate a small program with several instructions and determine how it goes through the pipeline so that the different stages of the instructions are executed in parallel. Study the following code.

```
1  #include <iregdef.h>
2
3  .set noreorder
4  .text
5  .globl start
6
7  .ent start
8  start:
9          lui     $9, 0x8002   # Load upper half of port address
10                              # Lower half is filled with zeros
11 repeat:
12         lbu     $8, 0x40($9) # Read from the input port
13         nop                  # Needed after load (is one NOP enough?)
14         sb      $8, 0x50($9) # Write to the output port
15         b       repeat       # Repeat the read and write cycle
16         nop                  # Needed after branch (is one NOP enough?)
17         li      $8, 0        # Clear the register
18 .end start
```

Build and upload the program above to the pipeline simulator and execute it step by step. Carefully note when the instructions are launched and when their results are ready. Also note how many instructions are in different stages of their execution at the same time.

1.6.2 Experiment 2

Modify the following code by assigning distinct values to `t0`, `t1`, and `t3`, and then build the program and single step through the instructions in the simulator.

```
1  #include <iregdef.h>
2
3  .set noreorder
4  .text
5  .globl start
6
```

```
.ent start
start:
        add     t2, t0, t1
        add     t4, t2, t3
        nop
        nop
        nop
.end start
```

- After how many clock cycles will the destination register of the first add instruction (that is `t2`) receive the correct result value?
- After how many clock cycles is the value of `t2` needed in the second instruction?
- What is the problem here? What is this kind of hazard called?

1.6.3 Experiment 3

This kind of problem can be solved by code reordering, introduction of **NOP** instructions, stalling the pipeline (hazard detection), and by forwarding. Explain when the first three methods can be used and how they work.

The MIPS processor hardware and the simulator both use *forwarding* to solve problems associated with data hazards. So far, you have used a version of the pipeline (S-script) which does not have forwarding. Now we are going to utilize the pipeline in the Xl-script directory, which takes advantage of forwarding. Single step through the program above using the new pipeline and examine how forwarding works.

- How does the forwarding unit know that it should bypass a value to an earlier stage?
- Were are the origin and destination of the forwarded data in the example above?

1.6.4 Experiment 4

Switch back to the simple version of the pipeline without forwarding (S-script). Modify the following code by assigning the same value to `t0` and `t1`, and single step through the instructions.

```
#include <iregdef.h>

.set noreorder
.text
.globl start

.ent start
start:
        nop
        nop
        beq     t0, t1, start
        addi    t0, t0, 1
        nop
        nop
```

```
15          nop
16  .end start
```

- How many cycles does it take until the branch instruction is ready to jump? What has happened to the following `addi` instruction while the branch is being calculated?
- What is the problem here? What is this kind of hazard called?
- What are the possible solutions to this problem?
- Switch to the forwarding-enabled pipeline (Xl-script) again. How does this version handle `beq`?

1.6.5 Experiment 5

Run the following program on the pipeline simulator (simple version). Assign distinct values to `t0` and `t1`, and let `t2` contain the address to a memory location where you know its contents. Then single step through the instructions.

```
1   #include <iregdef.h>
2
3   .set noreorder
4   .text
5   .globl start
6
7   .ent start
8   start:
9           lw      t0, 0(t2)
10          add     t1, t1, t0
11          nop
12          nop
13          nop
14  .end start
```

- After how many clock cycles will the destination register of the load instruction, `t0`, receive the correct result value?
- After how many clock cycles is the value of `t0` needed in the `add` instruction?
- What is the problem here? What is this kind of hazard called? This kind of problem can be solved with forwarding or hazard detection and stalling, just as other data hazards, but most MIPS implementations do not have these for load.
- What are the alternative solutions that can be used?
- Does the forwarding version of the simulator (Xl-script) handle the problem with delayed load?

1.7 Instruction Level Parallelism

Fire up your browser and go to `http://dark.eit.lth.se/` and click on this link `http://dark.eit.lth.se/cgi-bin/ScoreBoard.pl` - works best with Internet Explorer)

In this section we move from compiler-driven to hardware-driven optimizations. We will look at the effect of using the *Scoreboarding* algorithm. This is done in order to replace the pre-ordering we did so far with out-of-order execution. The instructions are tracked in the following manner. First, one can see the status of the respective instructions while they flow through the pipeline. Secondly, the status of the functional units are shown in nine fields:

- FU: Operation to perform in the unit (e.g. `add` or `sub`)
- Status busy: Indicates whether the unit is busy or not
- Fi: Destination register name
- Fj and Fk: Source register names
- Qj and Qk: Name of functional unit producing the data for the source registers
- Rj and Rk: Flags indicating whether the source registers have received their data

And finally, the status of the registers can be observed. The history of the program execution is also available.

For each instruction record the clock cycle in which it enters a particular pipe stage.

1.7.1 Experiment 1

Run the following program and answer the questions:

```
1  ld      F6, 34(R2)
2  ld      F2, 45(R3)
3  multd   F0, F2, F4
4  subd    F8, F6, F2
5  divd    F10, F0, F6
6  addd    F6, F8, F2
```

- After how many clock cycles can this program branch back to the beginning? (that is, the first `ld` of the 2nd iteration can be issued)
- Does re-ordering influence the execution time of this program, and if yes how?
- Is there a Write-after-Read hazard present and how is it resolved?

1.7.2 Experiment 2

Now examine the following assembly sequence (use the first simulator at `http://dark.eit.lth.se/darklab/scoreboard/`).

```
1  ld      F0, 0(R1)
2  addd    F4, F0, F2
3  sd      F4, 0(R1)
4  ld      F0, -8(R1)
5  addd    F4, F0, F2
6  sd      F4, -8(R1)
```

- After how many clock cycles can this program branch back to the beginning?
- Does re-ordering influence the execution time of this program and how?
- Is there a Write-after-Read hazard present and how is it resolved?

1.7.3 Experiment 3

The last program that we will look at is the sum-of-products that appears in the Fast Fourier Transform.

```
1 ld      F0, 0(R1)
2 ld      F2, 4(R1)
3 multd   F8, F0, F2
4 ld      F4, 8(R1)
5 ld      F6, 10(R1)
6 multd   F10, F4, F6
7 addd    F10, F8, F10
8 ld      F8, 12(R1)
9 addd    F10, F10, F8
```

- After how many clock cycles can this program branch back to the beginning?
- Does re-ordering influence the execution time of this program and how?
- Is there a Write-after-Read hazard present and how is it addressed?

1.7.4 Experiment 4 - for interested students

Interested students can do the same experiments using the *Tomasulo* algorithm. Do you notice any difference?

1.8 Conclusions

Before you can get a pass on the laboratory exercise, think about the questions below and explain to your TA:

- How can a pipelined processor be faster than one without pipelining?
- What are the special problems that appear in pipelining?
- How can these problems be overcome?
- Is there a difference in writing compilers for pipelined processors?
- Which is faster, straight code or code with many branches?
- What does RISC mean? What does CISC mean?
- Is the Pentium processor pipelined? How about these: AMD Phenom? Intel Core2? ARM Cortex A?

2 Laboratory 2: Advanced pipelining

2.1 Goals

After this laboratory exercise, you should understand how program behavior (instruction class profiles) relates to branch prediction efficiency, as well as trade-offs related to their implementation. You should also have an understanding for the relative importance of various advanced pipeline techniques like branch prediction, variable pipeline width and out-of-order execution.

2.2 Literature

Hennessy and Patterson: Appendix A, Chapter 2-3
Section 5.3 of this lab-manual

2.3 Preparations

Read section 5.3 on simulation and the SimpleScalar tool-set thoroughly. You should be able to answer the home assignment questions. Read through this laboratory assignment and make sure that you have sufficiently familiarized yourselves with the required concepts in pipelining and branch prediction.

Note: In the course of these labs you will run a large number of simulations, and it may be difficult to keep track of your results unless you maintain a lab book (hard copy or on-line). This book should contain the description of all the simulation runs you performed, your simulation plans, comparison of results, graphs if any etc. In addition as you will start using more detailed simulators, the simulation time will increase. A lab book which documents all the simulation runs you performed already will help you avoid repeat runs, and will save you considerable time. The system allows you to attach an arbitrary ID to each run, as well as it saves all results within a session (typically one lab). Use this intelligently in order to keep better track of what you have done.

2.3.1 Home Assignment 1

Answer the following questions:

- What is the role of simulators in processor design?
- Why is it advantageous to have several different simulators?
- For the four branch prediction schemes '`taken|perfect|bimod|comb`', describe the predictor. Your description should include:
  - What information the predictor stores (if any)?
  - How the prediction is made?
- What is out-of-order execution?
- What is the difference between scoreboarding and Tomasulo?

2.4 Program behavior (instruction profiling)

Start a Web-browser (works best with Internet Explorer) and go to the initial lab-page (http://dark.eit.lth.se/) and login with your EFD-id. After the lab is finished and you have recorded all your measurements you should logout of the system.

When you have logged out all your results are unavailable so be sure to record them first!

Choose three of the available benchmarks from the '`Program to run`' menu (they are briefly described in section 5.3.3), and run the profiling simulator for each of them, to find out the distribution of instruction classes.

Fill table 1 with all available benchmark programs versus instruction class profiles. (Get values from other groups for those you didn't run yourself!)

benchmark	load	store	uncond branch	cond branch	integer computation	fp computation
anagram						
go						
compress						
applu						
mgrid						
swim						
perl						
gcc						

Table 1: Benchmark programs versus instruction class profiles in %

Choose one of the benchmarks for your further assignments based on the following considerations:

- Is your benchmark memory intensive or computation intensive?
- Is your benchmark mainly using integer or floating point?
- What percentage of the instructions executed are conditional branches? Given this percentage, how many instructions on average does the processor execute between each pair of conditional branch instructions (do not include the conditional branch instructions).
- Using your textbook, class notes, other references, and your own opinion, list and explain several reasons why the performance of a processor like the one simulated by sim-outorder (e.g., out-of-order-issue superscalar) will suffer because of conditional branches. For each reason also explain how, if at all, a branch predictor could help the situation.

2.5 Branch Predictors

2.5.1 Experiment 1

We will now use the branch prediction simulator (sim-bpred) to investigate the effects of branch predictors on the execution of your benchmark. This simulator allows you to simulate 5 different types of branch predictors. You can see the list of them by looking at the menu '`branch predictor type`' for the branch prediction simulator sim-bpred. (`bimod` is the 2-bit prediction scheme, figure 2.4, in the course-book. '2lev' is a two level adaptive branch predictor. 'comb'

combines a bimod and a 2-level predictor. The detailed simulator (sim-outorder) also implements the 'perfect' predictor which always make a correct prediction.)

For three of the possible branch prediction schemes, '`nottaken|taken|bimod`', run the simulation for your benchmark as you did above and note the branch prediction statistics for each in table 2.

benchmark	**nottaken**	**taken**	**bimod**
anagram			
go			
compress			
applu			
mgrid			
swim			
perl			
gcc			

Table 2: Branch prediction statistics

Note: the simulator statistics are for all branches - both conditional and unconditional (which are regarded as predicted correctly). For this reason the reported prediction rates for `taken` and `nottaken` do not add to 1. Use the branch-direction measurements and number of updates, both corrected for unconditional branches to calculate accuracy (*hit rate for conditional branches*).

2.5.2 Experiment 2

Use the detailed simulator (sim-outorder) to measure and describe how the prediction rate (bpred_dir_rate) effects the processor CPI (sim_CPI) for your benchmark. Also use this simulator to measure CPI when using the `perfect` branch predictor type. This simulator produces a lot of text (Simulation Statistics) as the result of a simulation, but you can use the browser search function on the result-page in order to find the desired result.

	taken		bimod		comb		perfect	
Benchmark	Branch pred. rate	CPI	Branch pred. rate	CPI	Branch pred. rate	CPI	Branch pred. rate	CPI

Table 3: Branch prediction rate versus CPI

There is a bug in the simulator, such that branch prediction rates for taken/nottaken are wrong. For the taken column only use the CPI from the simulation and use the rates from table 2!

(Again fill in values in table 3 for other benchmarks with the help of other groups.)

For the four branch prediction schemes '`taken|perfect|bimod|comb`', describe the predictor. Your description should include:

- What information the predictor stores (if any)?
- How the prediction is made?
- What the relative accuracy of the predictor is compared to the others.

2.6 Choosing a new branch strategy

Suppose you are choosing a new branch strategy for a processor. Your design choices are:

1. predict branches taken with a branch penalty of 2 cycles and a 1200 MHz clock-rate
2. predict branches taken with a branch penalty of 3 cycles and a 1300 MHz clock-rate
3. predict branches using bimod with a branch penalty of 4 cycles and a 900 MHz clock-rate
4. predict branches using bimod with a branch penalty of 4 cycles and a 1000 MHz clock-rate and half the L1 cache size

Hint - fill in table 4 for your chosen benchmark, with cycle count (sim_cycle) and CPI from detailed simulations. Calculate the execution time using the given clock-rate.

Benchmark		Alternative 1	Alternative 2	Alternative 3	Alternative 4
	Sim_cycle				
	CPI				
	Exe time				
	Sim_cycle				
	CPI				
	Exe time				
	Sim_cycle				
	CPI				
	Exe time				

Table 4: Impact of different branch strategies

Questions:

- What would be your choice for your benchmark? Why?
- How much do you have to be able to increase the clock frequency in order to gain performance when allowing a branch miss-prediction latency of 3 cycles instead of 2 when using the taken predictor?
- Compare your results with other groups using other benchmark programs and discuss your observations.

2.7 In-order vs out-of-order issue

Now you will conduct experiments to find out how the increase in the parallelism in processing instructions affects the CPI of your processor, and how you can improve the performance of memory reference instructions.

In all experiments you will use the default cache and branch predictor configurations.

2.7.1 Experiment 1

Experiment with in-order and out-of-order issue, and the width of the pipeline, by running the simulation with the following combinations of parameters. Measure CPI and total no of cycles. (**Note:** out-of order issue and execution is default. in-order issue is selected with the check-box labeled '`run pipeline with in-order issue`').

- Pipeline width 1, in-order and out-of-order issue
- Pipeline width 4, in-order and out-of-order issue
- Pipeline width 8, in order and out of order issue

Benchmark		Pipeline width=1		Pipeline width=4		Pipeline width=8	
		Sim_cycle	CPI	Sim_cycle	CPI	Sim_cycle	CPI
	Out-of-order						
	In-order						
	Out-of-order						
	In-order						

Table 5: In-order and out-of-order issue versus pipeline width

Questions:

- What is the impact on CPI of the increased pipeline width?
- Explain the impact and their difference for both in-order and out-of-order issue.

2.7.2 Experiment 2

Run the sim-outorder simulator varying the number of memory ports available: 1,2 and 4. Use a pipeline width of 4.

Benchmark		Memory port=1		Memory port=2		Memory port=4	
		Sim_cycle	CPI	Sim_cycle	CPI	Sim_cycle	CPI
anagram	Out-of-order						
anagram	In-order						
compress	Out-of-order						
compress	In-order						

Table 6: In-order and out-of-order issue versus memory ports

Questions

- What is the impact on CPI of the increase in available memory ports?

2.8 Conclusion

Before you pass the laboratory exercise, think about the questions below and explain to your supervisor:

- Why is bimodal branch prediction more expensive to implement than predict nottaken?

- Why is bimodal better than nottaken?
- What, if any, is the impact on CPI by allowing out-of-order issue?
- What, if any, is the impact on CPI by allowing more instructions to be processed in one cycle?
- Is the wider pipeline more effective with in-order or out-of-order issue, and if so - why?

3 Laboratory 3: Cache memory

3.1 Goals

A cache memory is a memory that is smaller but faster than the main memory. Due to the locality of memory references, the use of a cache memory can give the effect on the computer system that the apparent speed of the memory is that of the cache memory, while the size is that of the main memory. The actual efficiency gained by using a cache memory varies depending on cache size, block size, and other cache parameters, but it also depends on the program and data. In short, everything depends on a proper parametrization.

After this laboratory exercise, you should understand the basic principles of cache memories, and how different parameters of a cache memory affects the efficiency of a computer system.

3.2 Literature

Hennessy and Patterson: Appendix C, Chapter 5
MIPS Lab Environment Reference Manual (section A.2 in this lab-manual)
Cache tutorial at http://www.ecs.umass.edu/ece/koren/architecture/Cache/tutorial.html

3.3 Preparations

Use the cache tutorial to familiarize yourself with cache concepts and terminology.

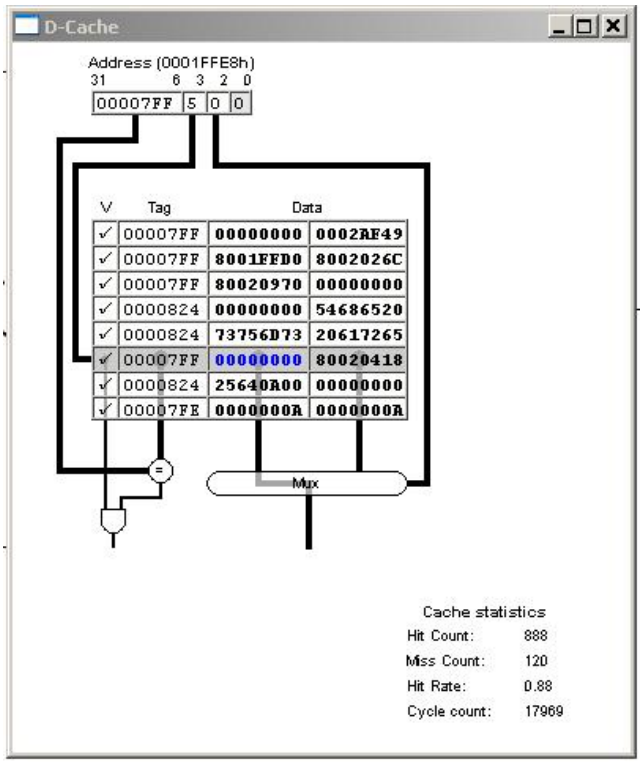

Figure 5: Data cache organization (see also HP fig C.5). (Note there is a bug in the cache simulator that makes the cache statistics counters wrap around at 100 000.)

Read the literature and this laboratory exercise in detail, and solve the home assignments. Note that you must solve the home assignments, or you will not be allowed to start the laboratory

exercise.

It is mandatory to be familiarized with cache concepts and terminology. As a first indication of being initiated on the subject, you should be able to solve conceptual issues like:

3.3.1 Home Assignment 1

Show the address bit partitioning for a memory system with
Memory size = 32MB
Cache size = 64 KB Block size = 16 Bytes
Set associative with 8 blocks per set. (What is a block?)

3.3.2 Home Assignment 2

In which order is the micro-operations done during a successful cache access (cf HP figure C.5).

- choose cache block
- valid address and divide into fields
- data sent to CPU
- compare tag with address field
- choose word within cache block

3.3.3 Home Assignment 3

Explain the following: cache size, block size, number of sets, write policy, and replacement policy.

3.3.4 Home Assignment 4

The following C program contains two subroutines which returns the sum of all the matrix cells. The only difference between the two subroutines is that they visit the matrix elements in a different order. This may seem unimportant, but with a cache memory, it may make a big difference.

Study the C-program carefully so that you understand in which order the matrix elements are used. The C-programs used here are available for downloading from the 'Software tools for Computer Architecture' page at http://dark.eit.lth.se/.

```
#include <stdio.h>
#include <idt_entrypt.h>
#define N 10

int A[N][N];

int SumByColRow (int Matrix[N][N])
{
  int i, j, Sum = 0, Time;

  flush_cache();
```

```
  timer_start();
  for (j = 0; j < N; j ++) {
    for (i = 0; i < N; i ++) {
      Sum += Matrix[i][j];
    }
  }
  Time = timer_stop();
  printf("SumByColRow time: %d\n", Time);
  return Sum;
}

int SumByRowCol (int Matrix[N][N])
{
  int i, j, Sum = 0, Time;

  flush_cache();
  timer_start();
  for (i = 0; i < N; i ++) {
    for (j = 0; j < N; j ++) {
      Sum += Matrix[i][j];
    }
  }
  Time = timer_stop();
  printf("SumByRowCol time: %d\n", Time);
  return Sum;
}

main ()
{
  int a, b;

  printf ("Laboratory Exercise, Home Assignment\n");
  // Run one of the cases below; comment out the other
  // Case 1
  a = SumByColRow (A);
  printf ("The sum is %d\n", a);
  // Case 2
  b = SumByRowCol (A);
  printf ("The sum is %d\n", b);
}
```

3.4 Missing and hitting in a cache

Create your project in the MIPS IDE, type in the above C program, build and upload it to the cache simulator (program Mips in the `mips` catalogue (S:)). Run the program in the cache simulator **with the default settings** and study how the instruction cache works. Fill in the first line of table 8.

	I-cache	**D-cache**	**Memory**
Default setting	cache size = 16 block size = 2 blocks in sets = 1	cache size = 16 block size = 2 blocks in sets = 1	read cycles = 50, write cycles = 50 writepolicy=WriteThrough write buffersize = 0 replacementpolicy=Random

Table 7: Default settings of the simulator (All penalty enabled).

Hint: Wall-clock time is smaller if I-cache and D-cache windows are closed.

Questions:

Give *full* answers!

- How is the full 32 bit address used in the cache memory?
- What happens when there is a cache miss?
- What happens when there is a cache hit?
- How large is the block size?
- What is the function of the tag?

3.5 Parametrization

The parameters of the cache memory can be changed to test the effects of different cases.

Run the program in the cache simulator with the settings in Table 8, and study how the instruction cache works. Record the cycles of cache miss/hit, and total cycles according to the table.

Questions:

- Investigate the effects of different parameter settings. Explain the following: cache size, block size, number of blocks in sets, write policy, and replacement policy.
- If a cache is large enough that all the code within a loop fits in the cache, how many cache misses will there be during the execution of the loop? Is this good or bad? How should the code look like that would benefit the most from a large block size?

Hint: If you don't see any clear difference - change the code to increase the measured difference.

Settings		I-cache Hit rate	D-cache Hit rate	Simulation time
All	SumByColRow			
default	SumByRowCol			
I-cache and D-cache cache size =32 Others: default	SumByColRow			
	SumByRowCol			
I-cache and D-cache blockSize =8 Others: default	SumByColRow			
	SumByRowCol			
I-cache and D-cache Number of blocks in sets =2 Others: default	SumByColRow			
	SumByRowCol			
D-cache writepolicy=WriteBack Others: default	SumByColRow			
	SumByRowCol			
D-cache replacementpolicy=FIFO Others: default	SumByColRow			
	SumByRowCol			

Table 8: Settings for parameter experimentation.

3.6 Optimized parametrization

Compile the C-program with the compiler option 'Optimization high'. Run the program in the in the cache simulator with the settings in Table 9, and study how the data cache works.

Questions:

- Study the subroutines SumByColRow and SumByRowCol . Explain carefully in what order the memory addresses are visited by the two subroutines.

- Execute the program and study how many cache hits the two subroutines have. Is there a difference? Why?

Settings		I-cache Hit rate	D-cache Hit rate	Simulation time
I-cache: **Disable Penalty(*)**	SumByColRow			
D-cache: **cache size=64** **block size= 16** **Others: Default**	SumByRowCol			

Table 9: Settings for optimization.

- Also speculate if you can detect in which order the matrix elements are located in physical memory.

3.7 Matrix multiplication

Create a new project with the following C program. Compile your code with the compiler option **Optimization high**.

```
#include <stdio.h>
#include <idt_entrypt.h>
#define N 10
int A[N][N];

int initMatrix (int Matrix[N][N])
{
  int i, j;
  for (i = 0; i < N; i ++) {
    for (j = 0; j < N; j ++) {
      Matrix[i][j] = i*N+j;
    }
  }
  return 0;
}

int SumOfProdByRowCol (int Matrix[N][N])
{
  int i, j, k, r, Sum = 0, Time;
  flush_cache();
  timer_start();
  for (i = 0; i < N; i ++) {
    for (j = 0; j < N; j ++) {
      r = 0;
      for (k = 0; k < N; k = k + 1)
        r = r + Matrix[i][k]*Matrix[k][j];
      Sum += r;
    }
  }
  Time = timer_stop();
  printf("SumOfProd time: %d\n", Time);
  return Sum;
}

int main ()
{
```

```
  int a;
  initMatrix(A);
  printf ("Laboratory Assignment Matrix multiplication\n");
  a = SumOfProdByRowCol(A);
  printf ("The sum of products is %d\n", a);
}
```

Introduce a blocking factor and change the program accordingly to localize the operations. Run the new version and explain the differences in performance. One of the sample codes like this:

```
#define B 4
#define min(X,Y) (X>Y?Y:X)

int SumOfProdByRowCol_Blockfactor (int Matrix[N][N])
{
  int i, j, k, r, Sum = 0, Time;
  int jj, kk;
  flush_cache();
  timer_start();
  for (jj = 0; jj < N; jj = jj + B)
    for (kk = 0; kk < N; kk = kk + B)
      for (i = 0; i < N; i ++) {
        for (j = jj; j < min(jj+B, N); j ++) {
          r = 0;
          for (k = kk; k < min(kk+B, N); k = k + 1)
            r = r + Matrix[i][k]*Matrix[k][j];
          Sum += r;
        }
      }
  Time = timer_stop();
  printf("SumOfProd time: %d\n", Time);
  return Sum;
}
```

Simulate both of the programs with the settings shown in table 10.

Settings		**D-cache Hit rate**	**Simulation time**
I-cache: Disable Penalty(*) D-cache: block size= 4 Others: Default	Without Blocking factor		
	With Blocking factor		

Table 10: Settings for matrix multiplication program.

Questions:

- What is the blocking factor and how does it work? Draw diagrams to illustrate how the two programs work!

- How many methods do you know to reduce the cache miss rate?

3.8 Reflections/Conclusion

You should now be able to converse on typical cache dimensioning problems, like:

- What is the general idea with cache memory?

- How does block size affect the efficiency of a cache?

- How fast is a cache memory and a DRAM memory in relation to each other?

- Does the optimal cache parameters depend on the program code?

- How can one select good cache parameters?

4 Laboratory 4: Advanced cache; Cache coherency

4.1 Goals

After this laboratory exercise, you should have deeper understanding of how various cache parameters affects performance. Finally, you should get an insight into the cache inconsistency problem in a multiprocessor system.

4.2 Literature

Hennessy and Patterson: Chapter 2, 5; Appendix A, C
Section 5.3 of this manual

4.3 Preparations

Read section 5.3 on simulation and the SimpleScalar tool-set thoroughly.

Make sure that you have sufficiently mastered the concepts in chapters 2, 5 and appendixes A, C of Hennessy and Patterson. You can test some of this by answering the following home assignments.

Note: In the course of these lab you will run a large number of simulations, and it may be difficult to keep track of your results unless you maintain a lab book (hard copy or on-line). This book should contain the description of all the simulation runs you performed, your simulation plans, comparison of results, graphs if any etc. In addition as you will start using more detailed simulators, the simulation time will increase. A lab book which documents all the simulation runs you performed already will help you avoid repeat runs, and will save you considerable time. The system allows you to attach an arbitrary ID to each run, as well as it saves all results within a session (typically one lab). Use this intelligently in order to keep better track of what you have done.

4.3.1 Home Assignment 1

- What are the four main categories of cache performance optimizations? Relate these to the formula for average memory access time.

- Which of these categories does associativity affect?

- Which of these categories does block size affect?

4.3.2 Home Assignment 2

- How is associativity, number of blocks, number of sets and cache size related?

- How does these affect the average access time for L1- and l2-caches?

4.3.3 Home Assignment 3

- Describe the cache inconsistency problem and also some of the sources for this issue.

4.3.4 Home Assignment 4

- How does the MESI protocol work? Describe the corresponding state diagram in detail.

4.3.5 Home Assignment 5

- Can we solve the cache inconsistency problem by considering "write through" policy for all caches in the system?

4.4 Cache performance

Start a Web-browser and go to the initial lab-page (http://dark.eit.lth.se/) and login with your EFD-id. After the lab is finished and you have recorded all your measurements you should logout of the system.

When you have logged out all your results are unavailable so be sure to record them first!

Use a single run of sim-cheetah to simulate the performance of the following cache configurations for two different benchmarks.

- Unified cache (Reference stream to analyze)
- least-recently-used (LRU) replacement policy
- 16 to 1024 sets
- 1-way to 8-way associativity
- 32-byte cache blocks

Note: sim-cheetah provides results for a continuous range of associativity, in this case 1, 2, 3, 4, 5, 6, 7 and 8. In your analysis of cache behavior ignore the measurements for associativity which is not a power of two, ie. consider only associativity of 1, 2, 4 and 8.

- Using the output from sim-cheetah, for caches of equivalent size, verify if increasing associativity or the number of sets in the cache gives the most benefit. To do so **produce graphs** showing changes in miss rate as associativity/no of sets changes. Matlab routines for producing graphs (see section 5.4) can be downloaded from the 'Software tools for Computer Architecture' page at http://dark.eit.lth.se/.
- Repeat this for data only cache, and for instructions only cache ('`Reference stream to analyze`').

4.4.1 Relation block size, miss ratio, and mean access time

Run simulations <sim-outorder> for two different benchmarks with the following configurations:

- unified L1 cache with a size of 32 KB, associativity 2 and block sizes 16, 32, 64, 128, 256 bytes.
- The L2 data cache should be a fixed configuration with a total size of 512 KB and a block-size of at least 256 (choose reasonable parameters).
- Keep other parameters as default.

Note: Remember to set all the cache parameters (program, L2 data cache, unified L1 and L2 caches) *for each simulation.* Record data in table 11 (Hint - hit times for L1 and L2 (cache:dl1lat, cache:dl2lat) are given in the simulation statistics. Since L2 configuration is not changed during your experiment you can estimate a fixed number for L2 miss penalty using L2 block-size, memory latency and memory access bus width.)

L2 data cache			
fixed total size	No of sets	block size (≥ 256)	associativity
512 KB			

L1 data cache						Miss Rates	
L1 size	L1 assoc	No of sets	block size	CPI	av. mem. access time	MR-L1	MR-L2
32 KB	2		16				
32 KB	2		32				
32 KB	2		64				
32 KB	2		128				
32 KB	2		256				

Table 11: L1 and L2 data cache

- Make plots that show block size vs CPI, and average memory access time vs block size. Matlab routines for producing plots (see section 5.4) can be downloaded from the 'Software tools for Computer Architecture' page at http://dark.eit.lth.se/.
- How does average access time vary with block size?

4.5 Cache Inconsistency

Modern computers are designed based on multi processors with a shared memory. In such computer architectures, each processor has its own separate cache, which is called local cache, as shown in Fig. 6. During the system operation, some of these processors request a particular memory block and perform some processing on the cached block. Then, the modified cached

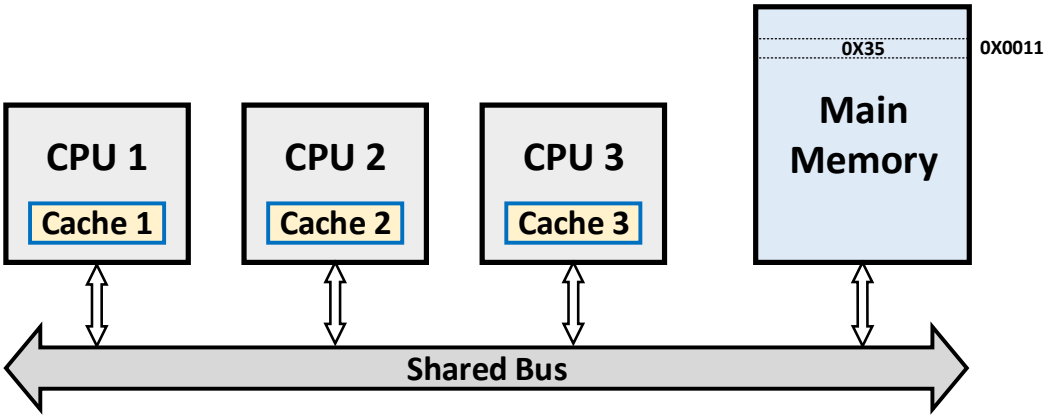

Figure 6: An example of multi processor architecture.

data will be saved in the corresponding local caches. So, in this case it is possible to have many different copies of that memory block: one copy in the main memory and one copy with different value in the local cache of the processor that requested it. So, these local modifications results in an *inconsistent view* of the main memory. Because, the change in one of these copies of data is not reflected by the other copies.
The main sources of inconsistency are:

1) Shared data:
Consider that the value of **A=0X35** is stored in address **0X0011** of the main memory in Fig. 6. CPU1 and CPU2 read the value of **A=0X35**. After some processing, CPU2 writes the value of **0X62** to **A** in its local cache. But, CPU1 will be left with an invalid cache of memory without any notification of that change. After that, CPU3 wants to read the content of address **0X0011** of the main memory. Depending on the write policy, the value of **A** in the main memory can be **A=0X35** or **A=0X62**. Cache coherence will manage such conflicts by maintaining a coherent view of the data values in multiple caches.

2) Process migration
In multiprocessor systems, usually the heavy tasks are distributed between multiple CPUs, depending on their current processing load. Consider that a certain task is running on CPU1 and system decides to move a part of this processing to another CPU, which has less load. In such case, the local caches of these two CPUs are not consistent (regardless of updating the content of the main memory).

3) I/O activity
When new data enters to the shared main memory, there is a potential of inconsistency between local caches and the main memory. Because, the input operations write something into the memory, while the content of local caches are not updated. Also, there is similar problem in writing data to an I/O while the sources of that data are not consistent.

4.6 Cache Coherency

There are many protocols to prevent from above cache inconsistency problems, which are called *Cache Coherence* protocols. Cache coherence is a mechanism that provides a uniform state for each cached block of data and it ensures that all the changes in the content of a shared data are

propagated throughout the whole system in the proper time.

4.6.1 MESI Protocol

MESI is one of the most common cache coherence protocols, which supports write-back caches. In this part of the lab, you will work with the *VivioJS* animation to understand the MESI cache coherency protocol.
Click **here** to open a web-browser and access to the VivioJS animation. Also, you can read more about the MESI protocol in the same page (*https://www.scss.tcd.ie/ Jeremy.Jones/vivio/caches/ MESIHelp.htm*). This animation implements three different scenarios. You should be able to perform the following tasks with VivioJS in this lab:

4.6.2 Scenario 1 – Bug Free

- Set the animation mode to "bug free!". Run the instructions in Table 12 by pressing the proper buttons of CPUs.
- Look at the direction of the traffic on the address and data busses after running each instruction. Then, write the important actions that you observed in the caches and memory, in the third column of Table 12. You should specify the type of operation (BR, BW, PR, PW), if caches or main memory are updated, and if that operation is done in the shared or non-shared mode.
- After running each instruction, fill in the corresponding cells in the last three columns of Table 12 (i.e. 6 cells are related to all 6 cache lines). In each cell, write a phrase in the format of "X-Y-Z", where "X" is the state of that cache line in the MESI FSM, "Y" is the address, and "Z" is the corresponding data. Results of the first instruction are written in Table 12 as an example. Write "U" if the content of that cache line is unknown.

4.6.3 Scenario 2 – Bug 0

It is possible to introduce bugs into the VivioJS animation by pressing the "bug free" button.

- Set the animation mode to "bug 0". Run the instructions in Table 13 by pressing the proper buttons of CPUs.
- Fill in Table 13 as described above.
- Indicate that what the bugs are in this experiment. You should be able to explain all the transitions in both scenarios and relate them to the MESI FSM.

Conclusions

Before you pass the laboratory exercise, think about the questions below and explain to your supervisor:

- What is the relative gain for the various performance enhancements techniques used in this lab?

	Operation	Results of the Operation	Cache Status		
			Cache 0	Cache 1	Cache 2
1	CPU0: read a1	CPU0 reads a1 from the memory (not shared)	I-U-U E-a1-0	I-U-U I-U-U	I-U-U I-U-U
2	CPU1: write a1				
3	CPU2: write a1				
4	CPU0: read a1				
5	CPU1: read a0				
6	CPU0: write a0				
7	CPU1: write a2				
8	CPU2: read a0				
9	CPU0: write a2				
10	CPU2: read a2				
11	CPU1: write a1				
12	CPU1: read a3				
13	CPU2: read a1				

Table 12: The result of running a program in VivioJS animation -"bug free!" mode.

- Are the techniques investigated in this laboratory (cache organization, pipeline modifications) independent?
- How does MESI protocol help to solve the cache inconsistency problem?

References

[1] Europractice, UMC standard, *http://www.europractice-ic.com/general_prices.php*, 2011.

	Operation	Results of the Operation	Cache Status		
			Cache 0	Cache 1	Cache 2
1	CPU0: read a1	CPU0 reads a1 from the memory (not shared)	I-U-U E-a1-0	I-U-U I-U-U	I-U-U I-U-U
2	CPU1: write a1				
3	CPU2: write a1				
4	CPU0: read a1				
5	CPU1: read a0				
6	CPU0: write a0				
7	CPU1: write a2				
8	CPU2: read a0				
9	CPU0: write a2				
10	CPU2: read a2				
11	CPU1: write a1				
12	CPU1: read a3				
13	CPU2: read a1				

Table 13: The result of running a program in VivioJS animation -"Bug 0" mode.

[2] ITRS, *http://www.itrs.net/Links/2001ITRS/PIDS.pdf*, 2001.

[3] ITRS, *http://www.itrs.net/Links/2003ITRS/PIDS2003.pdf*, 2003.

5 Appendix: Software

5.1 Pipeline simulator mipspipe2000.exe

See "Mips Lab Environment Reference Manual" section "Pipeline" available from 'Course Material' on the course Web-pages.

5.2 Cache simulator (mips.exe)

See "Mips Lab Environment Reference Manual" available available from 'Course Material' on the course Web-pages.

5.3 SimpleScalar simulator tool set

5.3.1 Getting Started with the SimpleScalar Tool Set

Based on the manual by Ewa Z. Bem, School of Computing and Information Technology, University of Western Sydney Nepean, which was based on the manual by Todd M. Bezenek, University of Wisconsin

Introduction

This document contains background material about the SimpleScalar toolset of simulators used in the Computer Architecture lab. SimpleScalar itself is available for download together with various tools and utilities including detailed documentation from http://www.simplescalar.com/

SimpleScalar and Simulation in Computer Architecture

When computer architecture researchers work to improve the performance of a computer system, they often use an existing system to simulate a proposed system. Although the intent is not always to measure raw performance (estimating power consumption is one alternative), performance estimation is one of the most important results obtained by simulation. The SimpleScalar tool set is designed to measure the performance of several parts of a superscalar processor and its memory hierarchy. This document describes the SimpleScalar simulators. Other simulation systems may be similar or very different.

Overview of SimpleScalar Simulation

The SimpleScalar tool set includes a compiler that creates binaries for a non-existent processor. The binaries can be executed on one of several simulators that are included in the tool set. This section describes the goals of processor simulation.

The execution of a processor can be modelled as a series of known states and the time (or other costs, ie., power) required to make the transition between each pair of states. The state information may include all or a subset of:

- The values stored in all memory locations.
- The values stored in and the status of all cache memories.
- The values stored in and the status of the translation-lookaside buffer (TLB).
- The values stored in and the status of the branch prediction table(s) or branch target buffer (BTB).

- All processor state (ie. the pipeline, execution units (integer ALU, load/store unit, etc.), register file, register update unit (RUU), etc.)

A good way to evaluate the performance of a program on a proposed processor architecture is to simulate the state of the architecture during the execution of the program. By simulating the states through which the processor will pass during the execution of a program and estimating the time (or other measurement) necessary for each state transition, the amount of time that the simulated processor will need to execute the program can be estimated.

The more state that is simulated, the longer a simulation will take. Complex simulations can execute 100s of times slower than a real processor. Therefore, simulating the execution of a program that would take an hour of CPU time on an existing processor can take a week on a complex simulator. For this reason, it is important to evaluate what measurements are desired and limit the simulation to only the state that is necessary to properly estimate those measurements. This is the reason for the inclusion of several different simulators in the SimpleScalar tool set.

Profiling

In addition to estimating the execution time of a program on the simulated processor, profile information may be of use to computer architects. Profile information is a count of the number or frequency of events that occur during the execution of a program. One common example of profile data is a count of how often each type of instruction (ie., branch, load, store, ALU operation, etc.) is executed during the running of a program.

Profile information can be used to gauge the relative importance of each part of a processor's implementation in determining its performance when executing the profiled program.

The SimpleScalar Base Processor

The SimpleScalar tool set is based on the MIPS R2000 processor's instruction set architecture (ISA). The processor is described in MIPS RISC Architecture by Gerry Kane, published by Prentice Hall, 1988. The ISA describes the instructions that the processor is capable of executing - and therefore the instructions that a compiler can generate - but it does not describe how the instructions are implemented. The implementation is what computer architects change in order to improve the performance of a processor.

An existing processor can be chosen as a base processor for several reasons. These may include:

- The architecture of the processor is well known and documented.
- The architecture of the processor is state-of-the-art and therefore it is likely to be useful as a base for the study of future processors.
- The architecture of the processor has been implemented as a real processor, allowing simulations to be compared to executions on a real, physical processor.

An important consideration in the choice of the MIPS architecture for the SimpleScalar tool set was the fact that the GNU GCC compiler was available in source-code form, and could compile to the MIPS architecture. This allowed the use of this public-domain software as part of the SimpleScalar tool set.

Description of the Simulators

The SimpleScalar tool set includes a number of simulators designed for various purposes. They are described below. For those simulators we are using there are also a description of the important profiling options available.

sim-bpred This simulator implements a branch predictor analyser.

sim-cache This simulator implements a functional cache simulator. Cache statistics are generated for a user-selected cache and TLB configuration, which may include up to two levels of instruction and data cache (with any levels unified), and one level of instruction and data TLBs. No timing information is generated.

sim-cheetah This program implements a functional simulator driver for Cheetah. Cheetah is a cache simulation package written by Rabin Sugumar and Santosh Abraham which can efficiently simulate multiple cache configurations in a single run of a program. Specifically, Cheetah can simulate ranges of single level set-associative and fully-associative caches.

```
#-option <args>   # <default> # description
-refs    <string> #      data # reference stream to analyze, {none|inst|data|unified}
-R       <string> #       lru # replacement policy, i.e., lru or opt
-C       <string> #        sa # cache configuration, i.e., fa, sa, or dm
-a       <int>    #         7 # min number of sets (log base 2, line size for DM)
-b       <int>    #        14 # max number of sets (log base 2, line size for DM)
-l       <int>    #         4 # line size of the caches (log base 2)
-n       <int>    #         1 # max degree of associativity to analyze (log base 2)
-in      <int>    #       512 # cache size intervals at which miss ratio is shown
-M       <int>    #    524288 # maximum cache size of interest
-c       <int>    #        16 # size of cache (log base 2) for DM analysis
```

Note that 'line size' above is the same as block size. Most of the parameters above are give as log base 2 of the number, ie a line size of 16 bytes is given as '`-l 4`.

sim-fast This simulator implements a very fast functional simulator. This functional simulator implementation is much more difficult to digest than the simpler, cleaner sim-safe functional simulator. By default, this simulator performs no instruction error checking, as a result, any instruction errors will manifest as simulator execution errors, possibly causing sim-fast to execute incorrectly or dump core. Such is the price we pay for speed!!!!

sim-outorder This simulator implements a very detailed out-of-order issue superscalar processor with a two-level memory system and speculative execution support. This simulator is a performance simulator, tracking the latency of all pipeline operations.

```
# -option        <args>       # <default> # description
-fetch:ifqsize   <int>        #       4 # instruction fetch queue size (in insts)
-fetch:mplat     <int>        #       3 # extra branch mis-prediction latency
-bpred           <string>     #   bimod # branch predictor type
                                         # {nottaken|taken|perfect|bimod|2lev|comb}
-bpred:bimod     <int>        #    2048 # bimodal predictor config (<table size>)
-decode:width    <int>        #       4 # instruction decode B/W (insts/cycle)
```

```
-issue:width     <int>        #       4 # instruction issue B/W (insts/cycle)
-issue:inorder   <true|false> #   false # run pipeline with in-order issue
-issue:wrongpath <true|false> #    true # issue instructions down wrong execution paths
-commit:width    <int>        #       4 # instruction commit B/W (insts/cycle)
-cache:dl1       <string>     # dl1:128:32:4:l # l1 data cache config
-cache:dl1lat    <int>        #       1 # l1 data cache hit latency (in cycles)
-cache:dl2       <string>     # ul2:1024:64:4:l # l2 data cache config
-cache:dl2lat    <int>        #       6 # l2 data cache hit latency (in cycles)
-cache:il1       <string>     # il1:512:32:1:l # l1 inst cache config
-cache:il1lat    <int>        #       1 # l1 instruction cache hit latency (in cycles)
-cache:il2       <string>     #     dl2 # l2 instruction cache config
-cache:il2lat    <int>        #       6 # l2 instruction cache hit latency (in cycles)
-mem:lat         <int list...># 18 2 # memory access latency (<first_chunk> <inter_chunk>)
-mem:width       <int>        #       8 # memory access bus width (in bytes)
-tlb:itlb        <string>     # itlb:16:4096:4:l # instruction TLB config
-tlb:dtlb        <string>     # dtlb:32:4096:4:l # data TLB config
-tlb:lat         <int>        #      30 # inst/data TLB miss latency (in cycles)
-res:ialu        <int>        #       4 # total number of integer ALU's available
-res:imult       <int>        #       1 # total number of integer multiplier/dividers availabl
-res:memport     <int>        #       2 # total number of memory system ports available (to CP
-res:fpalu       <int>        #       4 # total number of floating point ALU's available
-res:fpmult      <int>        #       1 # total number of floating point multiplier/dividers a

The cache config parameter <config> has the following format:

  <name>:<nsets>:<bsize>:<assoc>:<repl>

  <name>   - name of the cache being defined
  <nsets>  - number of sets in the cache
  <bsize>  - block size of the cache
  <assoc>  - associativity of the cache
  <repl>   - block replacement strategy, 'l'-LRU, 'f'-FIFO, 'r'-random

  Examples:   -cache:dl1 dl1:4096:32:1:l
              -dtlb dtlb:128:4096:32:r

Cache levels can be unified by pointing a level of the instruction cache
hierarchy at the data cache hiearchy using the "dl1" and "dl2" cache
configuration arguments.  Most sensible combinations are supported, e.g.,

  A unified l2 cache (il2 is pointed at dl2):
    -cache:il1 il1:128:64:1:l -cache:il2 dl2
    -cache:dl1 dl1:256:32:1:l -cache:dl2 ul2:1024:64:2:l

  Or, a fully unified cache hierarchy (il1 pointed at dl1):
    -cache:il1 dl1
    -cache:dl1 ul1:256:32:1:l -cache:dl2 ul2:1024:64:2:l
```

sim-profile This simulator implements a functional simulator with profiling support.

```
# -option  <args>       # <default> # description
-nice      <int>        #         0 # simulator scheduling priority
-max:inst  <uint>       #         0 # maximum number of inst's to execute
-all       <true|false> #     false # enable all profile options
-iclass    <true|false> #     false # enable instruction class profiling
-iprof     <true|false> #     false # enable instruction profiling
-brprof    <true|false> #     false # enable branch instruction profiling
-amprof    <true|false> #     false # enable address mode profiling
-segprof   <true|false> #     false # enable load/store address segment profiling
-tsymprof  <true|false> #     false # enable text symbol profiling
-taddrprof <true|false> #     false # enable text address profiling
-dsymprof  <true|false> #     false # enable data symbol profiling
-internal  <true|false> #     false # include compiler-internal symbols during symbol profili
```

sim-safe This simulator implements a functional simulator. This functional simulator is the simplest, most user-friendly simulator in the simplescalar tool set. Unlike sim-fast, this functional simulator checks for all instruction errors, and the implementation is crafted for clarity rather than speed.

The sim-cache and sim-cheetah simulators simulate only the state of the memory system– they do not keep track of the timings of events. The sim-outorder simulator does. In fact, it simulates everything that happens in a superscalar processor pipeline, including out-of-order instruction issue, the latency of the different execution units, the effects of using a branch predictor, etc. Because of this, sim-outorder runs more slowly, but it also generates much more information about what happens in a processor.

Because sim-outorder keeps track of timing, it can report the number of clock cycles that are needed to execute the given program for the simulated processor with the given configuration.

5.3.2 Running simulation experiments with SimpleScalar

A Web user interface to run simple experiments using SimpleScalar simulators is available at http://dark.eit.lth.se/darklab/

It uses sessions based on a ID (eg your EFD-login) given as login ID at the start to be able to keep track of all the simulations done during the laboratory session.

The main screen (figure 7) allows you to set simulator specific options (defaults are filled in if appropriate), choose which program and which simulator to run. Only a selection of all options are available through this user interface. It also provides access to all results produced earlier in this session. Futhermore it provides functionality for compiling a few programs with a special version of GCC that produces code that the simulator can run.

Note: In the course of these labs you will run a large number of simulations, and it may be difficult to keep track of your results unless you maintain a lab book (hard copy or on-line). This book should contain the description of all the simulation runs you performed, your simulation plans, comparison of results, graphs if any etc. In addition as you will start using more detailed simulators, the simulation time will increase. A lab book which documents all the simulation runs you performed already will help you avoid repeat runs, and will save you considerable time. The system allows you to attach an arbitrary ID to each run, as well as it saves all results within a session (typically one lab). Use this intelligently in order to keep better track of what you have done.

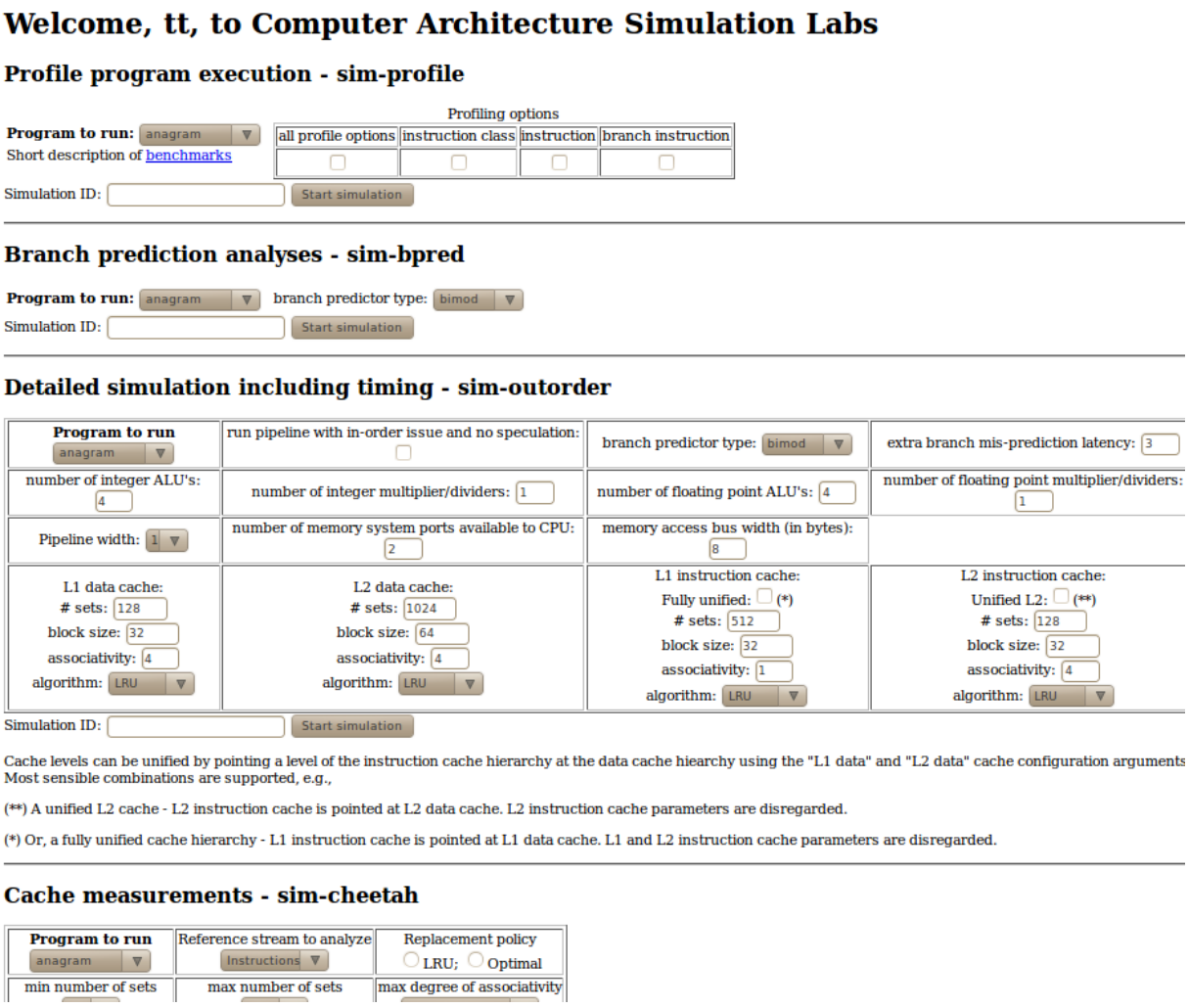

Figure 7: Main screen

5.3.3 Available benchmarks

There is more information available online linked from the main screen.

anagram A program for finding anagrams for a phrase, based on a dictionary.

compress (SPEC) Compresses and decompresses a file in memory.

go (SPEC) Artificial intelligence; plays the game of "Go" agianst itself

applu (SPEC) Parabolic/elliptic partial differential equations

mgrid (SPEC) Multi-grid solver in 3D potential field

swim (SPEC) Shallow water model with 1024x1024 grid

perl Calculates popularity of nodes in a graph based on the PageRank algorithm from Google.

gcc (SPEC) Limited version of GCC

5.3.4 Test programs for compilation

http://dark.eit.lth.se/darklab/anagram.txt
http://dark.eit.lth.se/darklab/stride.txt

5.4 MatLab routines for plotting results

These routines can be downloaded from the 'Software tools for Computer Architecture' page at http://dark.eit.lth.se/.

Use for Laboratory 4 assignment 'Cache performance':

```
set_assoc = [1, 2, 3, 4, 5, 6, 7, 8];
miss_rate_u = [ ]; % miss rates obtained for unified caches
miss_rate_i=[ ]; % miss rates obtained for instruction caches
miss_rate_d=[ ]; % miss rates obtained for data caches
%nbr_sets = [16, 32, 64, 128, 256, 512, 1024];

figure(1);
plot(set_assoc, miss_rate_u(1,:),'bd-',set_assoc, miss_rate_u(2,:), 'cs-', set_assoc,
 miss_rate_u(3,:), 'y^-',set_assoc, miss_rate_u(4,:), 'mx-',set_assoc, miss_rate_u(5,:),
'r+-',set_assoc, miss_rate_u(6,:), 'go-', set_assoc, miss_rate_u(7,:), 'kh-');
legend('16', '32', '64', '128', '256', '512', '1024');
grid on;
title('mgrid (Unified)'); % to be changed depending on the used benchmark
xlabel('Set Associativity (blocks/set)');
ylabel('Miss Rate');

figure(2);
plot(set_assoc, miss_rate_i(1,:),'bd-',set_assoc, miss_rate_i(2,:), 'cs-', set_assoc,
miss_rate_i(3,:), 'y^-',set_assoc, miss_rate_i(4,:), 'mx-',set_assoc, miss_rate_i(5,:),
'r+-',set_assoc, miss_rate_i(6,:), 'go-', set_assoc, miss_rate_i(7,:), 'kh-');
legend('16', '32', '64', '128', '256', '512', '1024');
grid on;
title('mgrid (Instruction)'); % to be changed depending on the used benchmark
xlabel('Set Associativity (blocks/set)');
ylabel('Miss Rate');

figure(3);
plot(set_assoc, miss_rate_d(1,:),'bd-',set_assoc, miss_rate_d(2,:), 'cs-', set_assoc,
miss_rate_d(3,:), 'y^-',set_assoc, miss_rate_d(4,:), 'mx-',set_assoc, miss_rate_d(5,:),
'r+-',set_assoc, miss_rate_d(6,:), 'go-', set_assoc, miss_rate_d(7,:), 'kh-');
legend('16', '32', '64', '128', '256', '512', '1024');
grid on;
title('mgrid (Data)'); % to be changed depending on the used benchmark
xlabel('Set Associativity (blocks/set)');
ylabel('Miss Rate');
```

Use for Laboratory 4 assignment 'Relation block size, miss ratio and mean access time':

```
ht_l1; % Hit Time for L1
```

```
ht_l2; % Hit Time for L2
mp_l2; % Miss Penalty for L2
block_size = [16, 32, 64, 128, 256];
cpi  = [];
mr_l1 = []; % Miss Rate for L1
mr_l2 = []; % Miss Rate for L2
AMAT = ht_l1 + mr_l1.*(ht_l2 + mr_l2 .* mp_l2);%Avarage Memory Access Time

figure(1);
plot(block_size, cpi,'bd-');
legend('L2 cache 128:128:4');% to be changed depending on the settings for L2
set(gca,'xtick',block_size);
title('applu'); % to be changed
grid on;
xlabel('Block size (Bytes)');
ylabel('CPI');

figure(2)
plot(block_size, AMAT,'bd-');
legend('L2 cache 128:128:4');% to be changed depending on the settings for L2
set(gca,'xtick',block_size);
title('applu');% to be changed
grid on;
xlabel('Block size (Bytes)');
ylabel('Average Memory Access Time');
```