CSE 30321 – Computer Architecture I – Fall 2011

Lab 01:	Architectural-level Performance Metrics	**Assigned:**	August 31, 2011
Total Points:	100 points	**Due:**	September 15, 2011

1. Goals

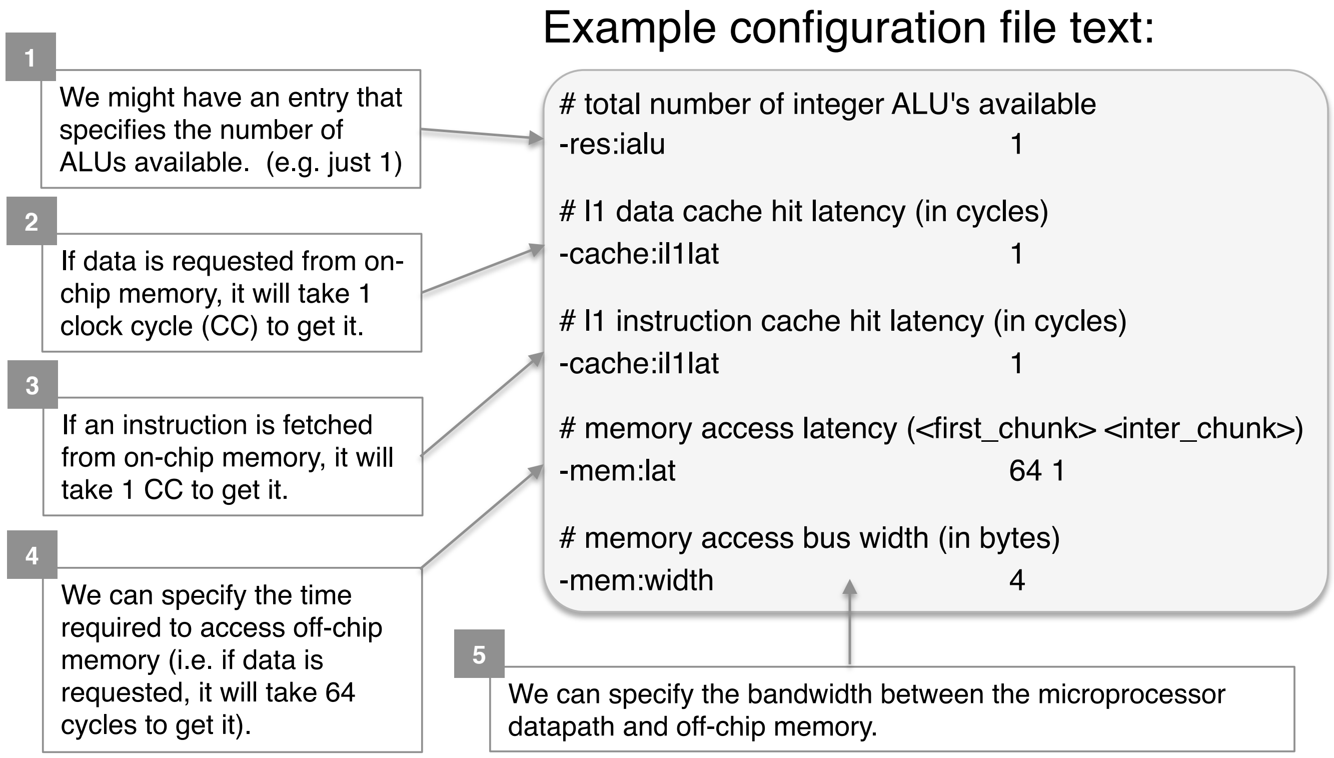
In this lab, we'll measure how hardware or software changes can impact program performance.

In lecture, we have discussed two important concepts – **(1)** Instruction Set Architectures (or ISAs) and **(2)** techniques for measuring architectural level performance. Upon completion of this lab, you should:

1. Have a better understanding as to what architectural level design features can help to improve performance – as well as when adding new features is *not* beneficial.
2. Even more importantly, you'll learn how to use simulation tools that can be leveraged to evaluate *any* potential architecture – and for different application spaces.

2. Introduction and Overview

We will use an architectural-level simulator called *SimpleScalar*. SimpleScalar allows you to describe a microprocessor's datapath and memory hierarchy in a simple, text-based configuration file.



Example configuration file text:

```
# total number of integer ALU's available
-res:ialu                           1

# l1 data cache hit latency (in cycles)
-cache:il1lat                       1

# l1 instruction cache hit latency (in cycles)
-cache:il1lat                       1

# memory access latency (<first_chunk> <inter_chunk>)
-mem:lat                            64 1

# memory access bus width (in bytes)
-mem:width                          4
```

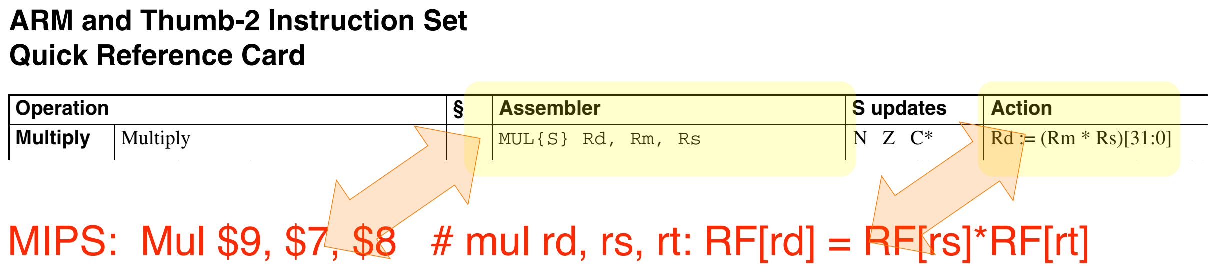
1
We might have an entry that specifies the number of ALUs available. (e.g. just 1)
2
If data is requested from on-chip memory, it will take 1 clock cycle (CC) to get it.
3
If an instruction is fetched from on-chip memory, it will take 1 CC to get it.
4
We can specify the time required to access off-chip memory (i.e. if data is requested, it will take 64 cycles to get it).
5
We can specify the bandwidth between the microprocessor datapath and off-chip memory.

(Regarding Items 2-5, recall from Lecture 1 that, generally, all of the data needed by a program will not be able to fit in the faster, on-chip memory. Instead, the most frequently used data is kept in faster, on-chip memory – and requests are made to slower, off-chip memory if data is not found on-chip.)

After describing the processor/memory architecture that you wish to evaluate, you can then use SimpleScalar to predict how a given *program* or *benchmark* will perform given that processor/memory configuration. Detailed, architectural-level simulations allow a computer architect (or compiler writer) to see whether or not design changes to a processor/memory configuration (or compiler) are beneficial at the application-level without actually building hardware.

<u>The version of SimpleScalar that we'll use is targeted toward the ISA associated with ARM microprocessors</u>. <u>There are several reasons for this choice</u>:

1. The ARM ISA is very similar to the MIPS ISA to be discussed in class, and is not dissimilar from the 6-instruction ISA for that matter. For example, as seen below, a multiply instruction for an ARM ISA essentially has the same syntax as a multiply instruction for the MIPS ISA.



ARM and Thumb-2 Instruction Set
Quick Reference Card

Operation		§	Assembler	S updates	Action
Multiply	Multiply		`MUL{S} Rd, Rm, Rs`	N Z C*	Rd := (Rm * Rs)[31:0]

MIPS: Mul \$9, \$7, \$8 # mul rd, rs, rt: RF[rd] = RF[rs]*RF[rt]

2. You should become comfortable with interpreting machine code for different microprocessor architectures.
    - A goal of this course is not to teach you MIPS or ARM assembly, but rather to (a) understand how the machine instructions for a RISC-like ISA accomplish some set of tasks specified by High-Level Language (HLL) code and (b) understand how changing code written in some HLL and/or a given processor-memory configuration can affect performance.

3. ARM microprocessors are used *everywhere*. Thus, the microprocessor configurations and benchmarks that we will work with in lab are very representative of hardware you use, and programs that you run everyday.

3. Benchmarks and processor configurations that we will study

3.1 <u>Benchmarks</u>:

<u>Different benchmark suites exist that allow a user to test a processor/memory configuration with a workload that is representative of how that processor/memory configuration might actually be used</u>.

For example, a common benchmark suite called ***SPEC*** is representative of what might be used for desktop-like workloads. Example programs in this suite include **gcc** (i.e. a compiler), **perl** (i.e. an interpreter), **vortex** (a database program), **gzip** (a compression tool), and **parser** (a grammar checker).

In this lab we'll use a benchmark suite called ***MiBench***[1]**.** MiBench is representative of functions that a microprocessor in an *embedded system* (a sensor, iPhone, iPod, e-Reader, etc.) might perform. The MiBench suite itself contains 35 benchmarks in 6 different classes – but we just concentrate on 3 of the benchmarks. Each benchmark will consist of a small dataset input and a larger dataset input (i.e. a small MP3 to encode and a large MP3 to encode).

The 3 benchmarks that we will work with in this lab are summarized on the next page.

[1] MiBench: A free, commercially representative embedded benchmark suite," M.R. Guthas, et. al.

Ispell

Benchmark class

The **Office** applications are primarily text manipulation algorithms to represent office machinery like printers, fax machines and word processors. The PDA market mentioned in the Consumer category also relies heavily on the manipulation of text for data organization.

Benchmark description

Ispell is a fast spelling checker that is similar to the Unix spell, but faster. It supports contextual spell checking, correction suggestions, and languages other than English. The input consists of a small and large document from web pages.

What to look for…

In the output text file, you will see suggestions for words that are misspelled, mis-hyphenated, etc. For example, look for the word “instrment” when examining the benchmark output for the small input file.

Lame

Benchmark Class

The **Consumer Devices** benchmarks are intended to represent the many consumer devices that have grown in popularity during recent years like scanners, digital cameras and Personal Digital Assistants (PDAs). The category focuses primarily on multimedia applications with the representative algorithms being jpeg encoding/decoding, image color format conversion, image dithering, color palette reduction, MP3 encode/decoding, and HTML typesetting.

Benchmark description

Lame is an MP3 encoder that supports constant, average and variable bit-rate encoding. It uses small and large wave files for its data inputs.

What to look for…

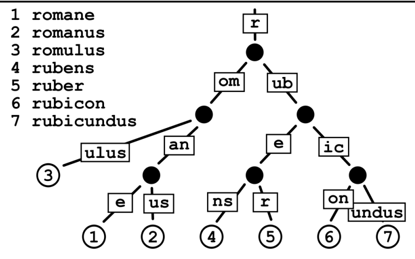
You should be able to play the output MP3 file in a standard MP3 player. You can also use the output of this benchmark as an input to the MP3 decode benchmark in MiBench (mad) – although this is not required. Note that this benchmark will probably take the longest to run.

Patricia

Benchmark Class

The **Network** category represents embedded processors in network devices like switches and routers. The work done by these embedded processors involves shortest path calculations, tree and table lookups and data input/output.

Benchmark description



A *Patricia* trie is a data structure used in place of full trees with very sparse leaf nodes. Branches with only a single leaf are collapsed upwards in the trie to reduce traversal time at the expense of code complexity. Often, Patricia tries are used to represent routing tables in network applications. The input data for this benchmark is a list of IP traffic from a highly active web server for a 2 hour period. The IP numbers are disguised. See example of text Patricia trie in this box.

What to look for…

The acronym PATRICIA stands for: “**P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric". When you run this benchmark, there will be a search for the IP addresses in the input file. When each is found, this will be noted as output.

3.2 Processor Configurations:

In this lab, you will use two Simplescalar configuration files that describe two commercial ARM microprocessors: the ***StrongARM*** and the ***XScale*** (the successor to the StrongARM). StrongARM chips were/are used in cell phones, PDAs, etc. Similarly, various versions of the XScale core are used in various Blackberry devices from Research in Motion, Amazon's Kindle e-reader, etc.

A brief comparison of the StrongARM architecture to the XScale architecture is given in Table 1.

Table 1: Comparison of StrongARM to XScale processor.

Design Parameter	StrongARM	XScale	Comment
# of Integer ALUs	1	1	
# of Integer Multiplier/Dividers	1	1	
# of Floating Point ALUs	1	1	
# of Floating Point Multiplier/Dividers	1	1	
Amount of on-chip memory	16 KBytes	32 Kbytes	The XScale has 2X the amount of faster, on-chip memory
Time required to access on-chip memory	1 CC	1 CC	The time required to access faster, on-chip memory is the same on the StrongARM and the XScale
Time required to access off-chip memory	64 CCs	32 CCs	The time required to access off-chip memory is 50% lower on the XScale
Bandwidth between microprocessor and off-chip memory	32 Bits	64 Bits	The bandwidth between the microprocessor and off-chip memory is 2X higher with the XScale

Note that unless you attempt to answer an extra credit question or wish to change the name of the file that simulation output is written to, you will not need to edit the configuration files for this lab as they are included with the benchmark directory hierarchy that you will copy over from the course AFS space.

4. How to setup & use the SimpleScalar / Mibench simulation environment

4.1 Preparing to run the Mibench suite with SimpleScalar:

1. The SimpleScalar executable will need to be run on a linux machine. For a list of machines that you should be able to log in to and use, see Appendix A. Thus, to get started:
    - Using a terminal program, **log on to a linux machine**.

2. Copy the directory listed below (from the course directory) into your home directory.
    - /afs/nd.edu/coursefa.11/cse/cse30321.01/Labs/01/lab_benchmarks
    - The directory "lab_benchmarks" contains processor configuration files and scripts that are needed to run each benchmark. The scripts were included so that you do not have to spend an excessive amount of time determining what arguments a particular benchmark takes to run. The scripts will invoke the SimpleScalar simulation engine with the pre-compiled MiBench executables and any necessary benchmark input files.

3. Next, update your path to point to the SimpleScalar executable. The executable can be found at:
    - /afs/nd.edu/coursefa.11/cse/cse30321.01/arm/simplesim-arm

    - It is called "sim-outorder"

- To update your path (i.e. so that you can run the simulator just by typing "sim-outorder" from the command line) add the following line to your .cshrc file:

```
setenv PATH /afs/nd.edu/coursefa.11/cse/cse30321.01/arm/simplesim-arm:$PATH
```

- Additionally, we will also use an extension to SimpleScalar called "sim-panalyzer" that estimates the average power required to execute a given benchmark. To use this executable, you should update your path with the line.

```
setenv PATH /afs/nd.edu/coursefa.11/cse/cse30321.01/arm/sim-panalyzer/PA2.0.3/Implementations/ARM/sim-panalyzer-2.0:$PATH
```

4.2 Running a SimpleScalar simulation:

Benchmarks will ultimately be run with small and large datasets assuming a StrongArm and XScale datapath. For example, to use SimpleScalar to run the *ispell* benchmark on the StrongArm datapath with a large dataset, simply:

Go to the directory:	~/lab_bechmarks/ispell/large/sa1/
At the command prompt, type:	./ispell_large_sa1

The simulation will begin, and the output will be written to a text file. The name of the text file is specified in the first entry in the configuration file (i.e. sa1core_large.cfg in the directory above):

```
# redirect simulator output to file (non-interactive only)
-redir:sim          sa1core_large.txt
```

- Thus, if you want the simulation output to go do a different file, just change this file name.

Note that each simulation will take ~1-30 minutes to run.

4.3 Preparing to use the SimpleScalar cross compiler:

1. In this lab, we will also use the SimpleScalar cross compiler – which allows you to compile C-code into ARM assembly such that it can be analyzed within the SimpleScalar environment.

   In other words, you can also use this tool to see how your own code's performance might be impacted by an architectural change and/or see the benefit of specific coding techniques for more efficient execution.

   - Again, it will be helpful if you update your path to include the *arm-gcc* executable.
   - To update your path (i.e. so that you can just type "arm-gcc" from the command line) add the following line to the path in your .cshrc file:

```
setenv PATH /afs/nd.edu/coursefa.11/cse/cse30321.01/arm/cross-compiler/gcc-4.1.1-glibc-2.3.2/arm-unknown-linux-gnu/bin:$PATH
```

After all path updates have been made, be sure to type "source .cshrc"
type "which sim-outorder" (for example) to check to see if this was done correctly

2. Thus, to compile any C-code so that it can be studied within the SimpleScalar environment, you can simply type:

   > arm-gcc foo.c –o foo –static (*must use static flag; libraries cannot be dynamically linked*)

3. To run this program in the SimpleScalar environment you can simply type:

   > sim-outorder –config sa1core.cfg foo

4.4 Analyzing the results of a SimpleScalar simulation:

- When you look at the simulation results (i.e. sa1core_large.txt) you will see LOTS of data. Some data will be more understandable later in the semester, while other data would make more sense after you take Computer Architecture II. However, there should be several lines in the file that are quite familiar. For example:

sim_num_insn	1588563478	# total number of instructions committed
sim_num_loads	259741195	# total number of loads committed
sim_num_stores	184385892	# total number of stores committed
sim_num_branches	147474085	# total number of branches committed
sim_CPI	1.5260	# cycles per instruction

- You may also want to pay attention to a few other lines too. For example:

dl1.accesses	435160624	# total number of [data] accesses
dl1.hits	435071297	# total number of hits
dl1.misses	89327	# total number of misses
dl1.miss_rate	0.0002	# [data] miss rate (i.e., misses/ref)
il1.accesses	1265769872	# total number of [instruction] accesses
il1.hits	1265384028	# total number of [instruction] hits
il1.misses	385844	# total number of [instruction] misses
il1.miss_rate	0.0003	# [instruction] miss rate (i.e., misses/ref)

  For example, the first 4 data items tell you how often data for a load or store instruction is found In “fast” memory. If there is a “miss”, then we must look for data in a “slower” level of memory. The “miss_rate” tells you how frequently a slower memory access is required. The second 4 items tell you how often an instruction encoding is found in fast memory (i.e. a “fetch”).

  Also, in the directory: /afs/nd.edu/coursefa.11/cse/cse30321.01/Labs/01/ there is a simple script – parse_Stats – that you can use to parse an output text file to extract all of the information / data that you will need to answer the questions in this lab

  - (If you change the output file naming convention, you may need to modify this slightly.)

Problem A

Question A.1:

Run the MiBench benchmarks discussed above assuming a StrongARM configuration and an XScale ARM configuration. Using simulation output from each benchmark, complete Table 2 below. Assuming that processor clock rates are the same, how does the more sophisticated (XScale) design affect speedup? (Try to comment on the suite as a whole, not just benchmark by benchmark.)

Table 2: Performance of benchmarks for small and large data sets given different configurations.

	StrongARM small input	**XScale small input**	**Speedup**	**StrongARM large input**	**XScale large input**	**Speedup**
Ispell						
Lame						
Patricia						

Question A.2

What if the clock rates for the StrongARM configuration and XScale configuration are different – i.e. the clock rate for the StrongARM is X and the clock rate for the XScale is Y? Is speedup affected? If so, how? (Again, try to comment on the suite as a whole, not just benchmark-by-benchmark.)

Question A.3

Do any benchmarks seem to especially benefit from the more sophisticated XScale configuration? If so, why do you think this is the case? And if so, try to support your conclusion with simulation data. (You might reference the discussion that compares and contrasts the StrongARM and XScale configurations. You might also reference some of the other data metrics that I discussed above.)

A.Extra Credit

For the benchmark where there is the biggest performance gap between the StrongARM design and the XScale design, create a hybrid configuration file and re-run the benchmark to see if there is a significant change if you alter the StrongARM architecture. How is performance affected?

Problem B

In this problem, you should begin to see how the way you write code in a HLL can significantly impact the time it takes for a microprocessor to actually run it – and in some instances, very inefficient looking code may end up giving you *better* performance.

For all of the questions in this problem, use the same processor configuration file provided at:

/afs/nd.edu/coursefa.11/cse/cse30321.01/Labs/01/xscale.cfg

Using the skeleton provided (see path below) as a guide, write a simple C program with two for loops – the first of which runs for 1,000,000 iterations, and the second of which runs for 9,000,000 iterations.

/afs/nd.edu/coursefa.11/cse/cse30321.01/Labs/01/if_else.c

- Within each loop, you should write an if – else if – else statement that checks to see whether or not 1 of 10 different conditions is met.

- The test condition is randomly generated.

- In the first loop, the default case will NEVER be met. For the second loop, the random number is updated so that the default case is ALWAYS met. (This is just an easy way to mimic a situation where the default case is likely to occur 90% of the time, and another case will occur the other 10% of the time.)

- Note that in the skeleton code, the default case is checked first.

Question B.1

Compile your completed program with the SimpleScalar cross compiler and, using data from a SimpleScalar simulation, calculate the execution time assuming that the clock rate is 500 MHz.

Question B.2

Now, re-write your code such that default case is checked *last*. Again, compile your completed program with the SimpleScalar cross compiler and, using data from a SimpleScalar simulation, calculate the execution time assuming that the clock rate is 500 MHz.

Question B.3

Compare and contrast the execution times calculated in Questions B.1 and B.2. Using output from the simulation (again, you can leverage the parse_Stats script), explain any differences and why you believe one version outperforms the other.

Question B.4

Finally, re-write you code such that the if – else if – else code is replaced with a switch statement – but where the default case is still considered last. Again, compile your completed program with the SimpleScalar cross compiler and, using data from a SimpleScalar simulation, calculate the execution time assuming that the clock rate is 500 MHz. Comment on the performance of this code when compared to the versions studied in Questions B.1 and B.2. Explain any differences and why you believe one version outperforms the other. (One way to answer this question would be to think about how the if-else gets compiled when compared to the switch statement code.)

Problem C

This problem is very similar to Problem B – but we'll look at some slightly different code.

More specifically, the SAXPY loop is a common routine in linear algebra and a common fixture in benchmark suites. SAXPY stands for "**S**ingle-precision real **A**lpha **X** **P**lus **Y**" and is a combination of scalar multiplication and vector addition. Quite simply, the routine is as shown below – and is common in many applications – including scientific code.

```
for (i=0; i<N; i++)
     q[i] = A * x[i] + y[i];
```

In this problem, we will consider several ways in which this code might be re-written – and will analyze its performance with SimpleScalar. Again, for all of the questions in this problem, use the same processor configuration file provided at:

```
/afs/nd.edu/coursefa.11/cse/cse30321.01/Labs/01/xscale.cfg
```

Additionally, an initial version of this code is provided at:

```
/afs/nd.edu/coursefa.11/cse/cse30321.01/Labs/01/loop.c
```

Question C.1

Compile the initial version of SAXPY with the SimpleScalar cross compiler and, using data from a SimpleScalar simulation, calculate the execution time assuming that the clock rate is 500 MHz.

Question C.2

We will now apply a technique called *loop fusion*. Simply put, we will combine loops. To understand how loop fusion can impact performance, combine the first two for loops in the initial version of SAXPY into one, recompile this version of SAXPY with the SimpleScalar cross compiler and, using data from a SimpleScalar simulation, calculate the execution time assuming that the clock rate is 500 MHz.

Question C.3

Next, we will apply a technique called *loop unrolling.* When a loop is "unrolled", N identical copies of the loop body are placed within the loop, and the loop counter is updated accordingly. Unroll the two loops in the version of SAXPY that you generated for Question C.2 so that 10 initializations and scalar multiplication / vector addition are done during each pass. Recompile this version of SAXPY with the SimpleScalar cross compiler and, using data from a SimpleScalar simulation, calculate the execution time assuming that the clock rate is 500 MHz.

Question C.4

Compare and contrast the 3 versions of SAXPY. Which performs best? Why? Support your answer with SimpleScalar simulation output.

Problem D

It is important to understand that better performance may not mean "faster execution time / lower latency." For example, one architecture may be considered advantageous over another if it can offer comparable – yet slower – execution time, and if battery life is significantly improved.

In the last part of this lab, you should rerun the *lame* benchmark (with small data sets) assuming the StrongARM and XScale ARM configurations. To run the benchmarks, you can use the same scripts / files that were used in Problem A. However, the "sim-panalyzer" executable should be used instead of "sim-outorder".

Record the average power.

- To find this in the output text file, type: "more <output.txt> l grep 'uarch.avgpdissipation'
- (Note that the units of the number provided are Watts)

Question D.1

Assume that for both configurations the microprocessor's clock rate is 233MHz. Calculate the execution time for the lame benchmark. Sim-panalyzer will call a slightly different version of sim-outorder, therefore the CPI values between Problems A and C may not be exactly the same.

Question D.2

Compare the differences in execution time to the average amount of power required for each benchmark given each configuration. From the standpoint of battery life, do you think that the more sophisticated design is worthwhile for these two benchmarks?

What to turn in:

You should complete a ***typed*** report with answers to:

- Questions A.1 – A.3
- Questions B.1 – B.4
- Questions C.1 – C.4
- Questions D.1 – D.2

Appendix:

Linux machines you can use

- student0{0,1,2,3}.cse.nd.edu
    - i.e. student 01@cse.nd.edu
- Also, see the list of machines at:
    - http://crcmedia.hpcc.nd.edu/wiki/index.php/Linux_Cluster_Host_List

How to keep benchmarks running after you've logged out…

- To learn how to keep benchmarks running after you have logged out, see the link on the course website next to the lab handout link.