# CS 6290: High-Performance Computer Architecture

# Project 0

This project is intended to help you set up and use the simulator for the other three projects, and also to help you understand the importance of branch prediction a little bit better. To complete this project, you will need to set up VirtualBox and the CS6290 Project virtual machine (VM).

You will put your answers in the red-ish boxes in this Word document, and then submit it in Canvas (the submitted file name should still be PRJ0.docx). You should not modify any other text in this document, and your answers should be in the red-ish boxes, in red letters and with a red underline. The font settings for the red-ish boxes are already set that way, so you should be OK if you just put your answers in and don’t change the font colors or underline settings. Georgia Tech students have full access to Office 365 so opening and editing of this document should not be a problem for anyone, and we will not accept answers in any other format. If you use another tool to edit and save the file, please make sure that it preserves at least the font colors and underline settings, otherwise graders might not find your answers easily, and points lost for that reason will stay lost.

In each answer box, you **must first provide your answer to the actual question** (e.g. a number). You can then use square brackets to provide any explanations that the question is not asking for but that you feel might help us grade your answer. E.g. answer 9.7102 may be entered as **9.7102 [Because 9.71+0.0002 is 9.7102].** For questions that are asking “why” and/or “explain”, the correct answer is one that concisely states the cause for what the question is describing, and also states what evidence you have for that. Guesswork, even when entirely correct, will only yield 50% of the points on such questions.

Additional files to upload are specified in each part of this document. **Do not archive** (zip, rar, or anything else) the files when you submit them – each file should be uploaded separately, with the file name specified in this assignment. You will lose up to 10 points for not following the file submission and naming guidelines, and if any files are missing you will lose all the points for answers that are in any way related to that file (yes, this means an automatic zero score if the PRJ0.docx file is missing). Furthermore, if it is not VERY clear which submitted file matches which requested file, we will treat the submission as missing that file. The same is true if you submit multiple files that appear to match the same requested file (e.g. several files with the same name). In short, if there is any ambiguity about which submitted file(s) should be used for grading, the grading will be done as if those ambiguous files were not submitted at all.

As explained in the course rules, **this is an individual project**: **no collaboration with other students or anyone else is allowed**.

### Part 1 [60 points]: Simulator set-up and simulation

Install the Oracle VirtualBox VMM (Virtual Machine Manager), then download the CS6290 Project virtual machine using the link from the Project 0 page in Canvas. DO NOT get the VM somewhere else, e.g. at the Udacity website, because those are likely older obsolete versions. Then import this VM into VirtualBox. Once you start (boot) the CS6290 Project VM from within VirtualBox, you should see a Linux desktop. Click on the topmost icon on the left-side ribbon and in the “Type Your Command” field type “Terminal”. Now you should get a terminal window – click on it and use it to issue commands we need.

Now, before you proceed to produce simulation results and edit files in the VM, it is very important for you to **regularly backup your files from the VM** to another location, e.g. your host machine, to avoid losing your work if the VM stops working. This may happen occasionally, e.g. in the past few semesters some students were not able to restart their VM's, which has caused them to lose work. The particular message that has been mentioned is "unreliable DMA position", and the only reliable solution to this is to use a fresh copy of the VM (which loses all the files created/modified inside the VM). So you should especially pay attention to making a **backup if you are going to restart/reboot your VM** (as opposed to just saving the VM state).

Now we are ready to get some real work done. The simulator in the “sesc” directory within our home directory, and we first need to build it. We will go into that directory and build the simulator:

cd ~/sesc

make

In the ~/sesc directory you should see (use ‘ls’) sesc.opt and sesc.debug links to executable files. We only compiled sesc.opt now, so only that link works, but that is enough for now because we will not be modifying the simulator in this project.

Now we need some application to run inside the simulator. The simulator simulates processors that use the MIPS ISA, so we need an application compiled for a MIPS-based Linux machine. Since these are hard to find, we have set up the Splash2 benchmark suite in the ‘apps’ subdirectory. For now, let’s focus on the lu application:

cd apps/Splash2/lu

There is a Makefile for building the application from its source code, which is in the Source subdirectory:

make

Now you should see (‘ls’) a lu.mipseb executable in the application’s directory. This is an executable for a big-endian MIPS processor – just the right thing for our simulator. So let’s simulate the execution of this program!

Well, the simulator can simulate all sorts of processors – you can change its branch predictor, the number of instructions to execute per cycle, etc. So for each simulation we need to tell it exactly what kind of system to simulate. This is specified in the configuration file. SESC already comes with a variety of configuration files (in the ~/sesc/confs/ directory), and we will use the one described in the cmp4-noc.conf file. So we use the following command line (this is all one line):

~/sesc/sesc.opt -fn256.rpt -c ~/sesc/confs/cmp4-noc.conf -olu.out -elu.err lu.mipseb -n256 -p1

Note that the dashes in the above command line are “minus” signs – pasting them into this document may have resulted in changing these to longer dashes, in which case you should change them back to “minus” signs. Also, note this is one command line that just does not fit in a single row in this document. Overall, be very careful if you copy-paste this from the document into the VM.

In this command line, the -f option tells the simulator to create a report file that ends with a dot and the specified string. The name of the report file is “sesc\_” followed by the name of the executable for the simulated application (lu.mipseb in this case), followed by a dot and the string we specify here. So, in this case sesc\_lu.mipseb.n256.rpt will be the file where the simulator stores simulation results. It is **very important** to delete the simulation file before we run a simulation that writes to the same file – if there is already a report file with the same name, the simulator appends its report to the file, and that causes the reporting scripts to combine the results of the simulations, and *that* means you will not get the results you need for the simulation you just ran!

The -c option tells the simulator which configuration file to use. The configuration file tells the simulator all sorts of parameters for the simulated processor, including what its branch predictor looks like, how big are its caches, etc. The -o and -e options tell it to save the standard and error output of the application into lu.out and lu.err files, respectively. This is needed because the simulator itself prints stuff, and we want to be able to look at the output and possible errors from the application to tell it executed OK. Note that there should be a space between “-c” and the configuration file name, but there should be no space between the “-o” and its file name, and there should be no space between “-e” and its file name.

Then the command line tells the simulator which (MIPS) executable to use, and what parameters to pass to that executable. The -n256 parameter tells the LU application (which does LU decomposition of a matrix) to work on a 256x256 matrix. The -p1 parameter tells the application to use only one core – we will want to use only one core until Project 3!

Now you should **look at the output of the application** – the lu.err file should be empty and the lu.out should have the “Total time without initialization: 0” line at the end. **If you do not check, you may get incorrect simulation results without knowing it**, e.g. because you had a typo in the command line that causes the application to abort without actually doing much. Applications not really finishing is a very common pitfall in benchmarking – we are running the applications just to see how the hardware performs, so we tend to not look at the output. However, if a run got aborted half-way through because of a problem (e.g. mistyped parameter, running out of memory, etc.), the hardware might look really fast for that run because execution time will be much shorter! So we need to check for each run whether it actually completed!

After the run completes, we also get a simulation report file (sesc\_lu.mipseb.n32 for the simulation we just ran). This report file contains the command line we used, a copy of the hardware configuration we used, and then lots of information about the simulated execution we have just completed. You can read the report file yourself, but since the processor we are simulating is very sophisticated, there are many statistics about its various components in the report file. To get the most relevant statistics out of the report file, you can use the report script that comes with SESC:

~/sesc/scripts/report.pl sesc\_lu.mipseb.n256.rpt

You can specify multiple files (or use wildcards), in which case the script prints the summary for each file you listed.

The printout of the report script gives us the command line that was used, then tells us how fast the simulation was going (Exe Speed), how much real time elapsed while the simulator was running (Exe Time), and how much simulated time on the simulated machine it took to run the benchmark (Sim Time). Note that we have simulated an execution that takes much less than a millisecond, but it probably took many milliseconds for the simulator to simulate that execution. Because simulation executes on one machine but is trying to model another, when reporting results there can be some confusion about whether we are referring to the execution of the simulator itself, or to the execution of the simulated application. To avoid this confusion, the simulation itself is called “native” execution and the simulated execution is sometimes also called “target” execution.

1. In this simulation, the simulated processor needed 54.240 milliseconds of simulated time (“Sim Time” in the report) to run the application, but the simulator itself took 45.580 seconds of native execution time to do this simulation (“Exe Time” in the report).

The printout of the report script also tells us how accurate the simulated processor’s branch predictor was overall (the number under Total). The next two numbers are in a parenthesis under the RAS label, and they report the accuracy of the return address stack (RAS) and what percentage of branches has actually used the RAS. Next, under the BPRed label we have the accuracy for branches that didn’t use the RAS (i.e. they used the direction predictor and possibly the BTB and/or the BTAC), and then we have statistics for the BTB and BTAC, which are less interesting for now.

1. The overall accuracy of the simulated branch predictor in this simulation was  
    94.72 percent. The RAS tends to have high accuracy, but it only took care of 2.74 percent of all dynamic branch instructions (i.e. branches as they occur at runtime). That means that the direction predictor handles the remaining \_\_\_\_\_97.26\_\_ percent of dynamic branch instructions and achieves \_\_\_\_\_94.57\_\_ percent accuracy.

Next, the report tells us how many dynamic instructions were completed (nInst) and the breakdown of these instructions, i.e. the percentage of nInst that comes from control flow instructions like branches, jumps, calls, returns, etc. (the number under the BJ label), from loads, from stores, from integer ALU, and from floating point ALU operations. The rest of that line (LD Forward and beyond) is less interesting for now.

1. The number of completed dynamic instructions in this run of lu on the simulated processor is 76250392 , and control flow instructions represent  
    9.44 percent of these instruction.

Next the report tells us the overall IPC achieved by the simulated processor on this application, how many cycles the processor spent trying to execute instructions, and what those cycles were spent on. This last part is broken down into issuing useful instructions (Busy), waiting for a load queue entry (LDQ) to become free, then the same for store queue (STQ), reservation stations (IWin), ROB entries, and physical registers (Regs, the simulator uses a Pentium 4-like separate physical register file for register renaming). Some of the cycles are wasted on TLB misses (TLB), on issuing wrong-path instructions due to branch mispredictions (MisBr), and some other more obscure reasons (ports, maxBr, Br4Clk, Other). Note that each of these “how the cycles were spent” items is a percentage although they do not have the percentage sign. Also note that the simulator models a multiple-issue processor, so the breakdown of where the cycles went is actually attributing partial cycles when needed. For example, in a four-issue processor we may have a cycle where one useful instruction is issued and then the processor could not issue another one because the LDW became full. In this cycle, 0.25 cycles would be counted as “Busy” and 0.75 cycles toward “LDQ”.

1. Given that branches represent 9.44 percent of all dynamically completed instructions, and that only 5.28 [100-94.72] percent of all completed branch instructions are mispredicted, we can conclude that only 0.498 percent of all dynamically completed instructions are mispredicted branches. But now we see that branch mispredictions result in wasting 9.2 percent of the processor’s performance potential. Why is this last number (percentage of performance wasted) so high compared to the previous one (percentage of instruction that are mispredicted branches)?

Branch mispredictions have a high wastage of cycles relative to other errors, even if they are only a small percentage of completed instructions. When a branch is mispredicted, the processor has to flush all the instructions that were speculatively executed based on the wrong branch prediction.

This means that several cycles of work are lost, and the processor has to start over with the correct instruction path, leading to a significant performance penalty. A deep pipeline with a late branch-resolution stage will have very large decreases in CPI when a branch miss occurs, due to the all the pipelined stages being flushed. Even in shallow pipelines, the branch misprediction will still flush many instructions, just less consequential than a deeper pipeline with more width/IPC.

Now run two additional simulations, this time using 16 and then 64 as the matrix size for lu (note the changes in -f and -n parameters):

~/sesc/sesc.opt -fn16.rpt -c ~/sesc/confs/cmp4-noc.conf -olu.out -elu.err lu.mipseb -n16 -p1

~/sesc/sesc.opt -fn64.rpt -c ~/sesc/confs/cmp4-noc.conf -olu.out -elu.err lu.mipseb -n64 -p1

Note: remember to check lu.err and lu.out after each simulation to make sure that the simulated run has completed correctly!

1. As we increase the matrix size (-n parameter) from 16 to 64, and then to 256, the overall branch predictor improves from 77.89% BPred to 94.29% BPred and then to 94.57% BPred percent. Why does this accuracy improve as we increase the –n parameter in lu?

BPred accuracy approves as we increase -n because as the matrix size increases, the loops within the program run more iterations, creating more consistent and predictable branch patterns. The improvement in accuracy is not because the predictor becomes better trained with more instructions, but because larger matrices contain more repetitive and predictable control flow code.

The mispredictions for all matrices will be the same or very similar, but the ratio of mispredictions to correct branch predictions will become higher for larger matrices due to the increased number of loop iterations, which are almost always predicted correctly aside from the end and beginning (depending on the predictor used and initial state). For example, a smaller matrix might have 2 mispredictions with 98 correct predictions, while a much larger matrix may have 2 or 3 mispredictions with 798 correct predictions.

Clearly, the larger matrix will be more accurate.

Note: Predictor accuracy is essentially the overall probability that an occurrence of branch instruction will be predicted correctly, i.e. the fact that we complete more instructions or that we complete more branches does not by itself explain why each branch occurrence is now more likely to be predicted correctly.

Note: We are not asking for guesswork - the answer to this question should include either *conclusive* evidence or *excellent* reasoning.

1. Submit all three simulator-generated report files (**sesc\_lu.mipseb.n16.rpt, sesc\_lu.mipseb.n64.rpt, and sesc\_lu.mipseb.n256.rpt**). Remember that each of these should be a result of only one simulation!

### Part 2 [40 points]: Compiling and simulating a small application

Now let’s see how to compile our own code so we can run it within the simulator. For that purpose, modify the hello.c in the ~/sesc/tests/ directory to print a greeting with your first and last name. The hello.c should be **exactly** as follows (e.g. do not omit the \n at the end of the string in the printf call, do not add extra spaces or other characters to the string, etc.), except that the name (the part in bold and cursive letters) should be your first and last name as show in Canvas:

#include <stdio.h>

int main(int argc, char \*argv[]){

printf("Hi! I am ***Milos Prvulovic***\n");

return 0;

}

Now you should try to compile and run your code natively, to see if it’s working:

gcc -o hello hello.c

./hello

This should result in printing out the “Hi! I am Milos Prvulovic” message and then a newline, with your own name of course. Once your code is working natively, you can try compiling it for the simulator. Remember that, because the simulator uses the MIPS ISA, we cannot use the native compiler. Instead, we use a **cross-compiler** – it is a compiler that runs on one machine (the native machine) but produces code for a different machine (the target machine, i.e. MIPS). Installing and setting up the cross-compiler is a time-consuming and tricky process, so we have already installed one and made it ready for you to use. It can be found in /mipsroot/cross-tools/bin with a name  
“mips-unknown-linux-gnu-gcc”. In fact, there a number of other cross-compilation tools there, all of them with a prefix “mips-unknown-linux-gnu-“, followed by the usual name of a GNU/GCC tool. Examples of these tools include gcc (a C compiler), g++ (a C++ compiler), gas (assembler), objdump (tool for displaying information about object files), addr2line (returns source file and line for a given instruction address), etc.

To compile our hello.c application, we can use the cross-compiler directly:

/mipsroot/cross-tools/bin/mips-unknown-linux-gnu-gcc -O0 -g -static -mabi=32 -fno-delayed-branch -fno-optimize-sibling-calls -msplit-addresses -march=mips4 -o hello.mipseb hello.c

We disable optimization (-O0), include debugging information in the executable file (-g), link with static libraries (-static) to avoid dynamic linking which requires additional setting up in the simulator, tell the compiler to avoid using branch delay slots  
(-fno-delayed-branch), regular call/return for all procedure calls  
(-fno-optimize-sibling-calls), use some optimizations that allow faster addressing  
(-msplit-addresses), and use a 32-bit implementation of the MIPS IV ISA (-mabi=32 -march=mips4).

After the compiler generates the MIPS executable file hello.mipseb for us, we can execute it in the simulator. Run this command line **exactly** as shown below – you should not change the command line options, run it from a different directory, or specify the path for hello.mipseb – that would change the simulated execution and prevent us from checking if you did it correctly and if everything is working.

~/sesc/sesc.opt -fhello0.rpt -c ~/sesc/confs/cmp4-noc.conf -ohello.out hello.mipseb

To complete this part of the project, get the simulation report summary (using report.pl) and fill in the blanks in the following statements:

1. In this simulation, the simulated processor executed 4868 (“nInst” in the report) instructions.
2. Why does it take so many instructions to execute even this small program?

Note: Again, answers based on guesswork will not earn points here!

When looking into the disassembly code and stripped code, the processor is executing a large amount of assembly instructions before and after reaching the main function. Routines like **\_start, \_libc\_start\_main, and <\_\_pthread\_initialize\_minimal>** are ran, which contain hundreds of instructions that are completed on top of just the main function. Many of these assembly lines setup and handle things that are abstracted into single lines of C code, such as return, resource cleanup, and mainly the C standard library / runtime.

When I ran the code with no printf() statement, nInst was 3952. This means that basically doing nothing costs almost 4000 instructions for the processor just to setup and cleanup a C program.

With my printf() function added back, the printf() increased the nInst count back to 4868 due to the loops and control flow.

Helpful Note #1: You can use mips-unknown-linux-gnu-objdump with a “-d” option to see the assembler listing of the hello.mipseb executable. In the listing produced by the disassembler, the code for the C “main” can be found by searching for “<main>”.

Helpful Note #2: When an executable file is loaded, it does *not* begin execution at main(). For what really happens, you can look up the documentation for the \_\_glibc\_start\_main function which is part of the GNU standard C library (GLIBC).

Helpful Note #3: Not all of the code you see in the listing from Hint 1 is actually executed. For example, you can use mips-unknown-linux-gnu-strip on the hello.mipseb executable to dramatically reduce its code size without breaking it (you can easily find online what the “strip” utility does). Even among the code that is not removed by “strip”, a lot of the code is only executed under special circumstances (failure to allocate memory, etc.) and is not reached in the specific run we are simulating.

1. Edit your hello.c to add an exclamation mark (“!”) to the string that is printed:

printf("Hi! I am Milos Prvulovic!\n");

then cross-compile and simulate your hello.mipseb program again (use -fhello1 to save the simulation report to a different file). Now the simulated processor has completed 4882 instructions. If the number of simulated instructions has increased, add another exclamation mark, compile, and simulate again with an incremented value for the -f option, so the report is saved to a new file with a number that matches the number of exclamation marks that were added. Keep doing this until the number of completed instructions goes down from the previous run. For example, if the number of simulated instructions increases when you add one exclamation mark, increases again when you add two, but with three you get fewer completed instructions than with two exclamation marks, then you should stop after the simulation with three exclamation marks. Explain why the number of executed (simulated) instruction decreases in this latest simulation compared to the previous one?

When looking at the disassembly code, we can notice that the printf() function is implemented within the main function through calling <\_IO\_puts>. Looking at the source code implementation, the decrease in instructions is due to the 3rd exclamation mark filling a buffer. If the string length matches or aligns with the buffer's size (every 4 bytes), **\_IO\_sputn** and **\_IO\_putc\_unlocked** can work more efficiently, avoiding additional instructions related to buffer management. This pattern repeats in the case of adding every 4 more characters, filling up the buffer in an optimal state.

This also coincides with the <strlen> call inside the \_IO\_puts function. Looking deeper into the source code, this function contains conditional branches that check every 4 bytes to determine where the null terminator for the string is. The branches will cause more instructions when these memory addresses are not aligned in the optimal way, such as odd bytes (strlen of 35 instead of 36 for example). So in return, we get less buffer handling needed and less control flow handling. Thus, there are overall less instructions.

Note: No guesswork or vague answers! The answer should be very specific, i.e. to answer this you need to point out specific behavior in specific code.

Helpful Note: You can use objdump to see what main is really doing (it’s **not** calling printf!) then find the source code for that function online instead of trying to figure out what it does by looking at the (barely human-readable) disassembly.

1. Submit your last **hello.c** source code (the one with the exclamation mark that made is execute fewer instructions than with one less exclamation), the original report file (**sesc\_hello.mipseb.hello0.rpt**), and the one from the last simulation (the one that resulted in a decrease in completed instructions, e.g. it **may** be **sesc\_hello.mipseb.hello3.rpt**).