Instructions:

1. Your deliverable for this project is a written report, consisting of a simulation log plus optional discussion. If your solution is near optimal, then only the simulation log is needed; otherwise, you need to submit a writeup discussing how you implemented this exercise. Detailed instructions are given at the end of the assignment. If submitted, please try to keep this writeup to 5 pages or less. Points will be deducted for inordinately long writeups.

2. Collaboration: You may work on this project only individually, i.e., not in teams. The work you hand in must be entirely your own.

Credits: This exercise was prepared by Cristian Soviani and Prof. Luca Carloni of Columbia University. Their contribution is gratefully acknowledged.

Preamble: This problem will introduce you to the SIMPLESCALAR simulator, which can be downloaded as simplesim-3v0d.tgz from http://www.simplescalar.com/ together with benchmark files and documentation. SIMPLESCALAR can be easily installed on the most common computer platforms. Do the following:

1. At http://www.simplescalar.com/ click on “Downloads/Tools” in the left panel, then select simplesim-3v0d.tgz.
2. Unpack simplesim-3v0d.tgz and follow the instructions in the “README” file in the directory simplesim-3.0/.
3. Download the simplescalar benchmarks archive file at http://www.eecs.umich.edu/mirv/benchmarks/supplied.tar.gz. This archive contains the benchmark go.ss that you will use for this problem.

Storyline: Your company wants to sell a computer which plays GO very fast. You are the CPU architect. You already have a CPU design with the following specs:

- clock frequency: 100 MHz
- core: 4-wide (i.e. 4-wide fetch, decode, issue, commit)
- integer units: 1 ALU and 1 MULT
- memory ports: 1
- fp units: none (the GO game does not use FP)
- 8kB 1-way instruction L1 cache (il1)
- 8kB 1-way data L1 cache (dl1)
- 64kB 2-way unified instruction/data/cache (ul2=il2=dl2)

Check-Point: At this point I assume that you can run sim-outorder on your computer and that you have the go.ss benchmark in your current working directory. If so, do the following:

1. Make an empty file called go.in.

1For more information on GO see www.en.wikipedia.org/wiki/Go_(board_game)
2. Then type the following command (broken down in multiple lines for clarity):

```bash
sim-outorder -redir:sim go.log \\
-cache:d1 dl1:256:32:1:1 \\
-cache:il1 il1:256:32:1:1 \\
-cache:d12 ul2:512:64:2:1 \\
-cache:il2 dl2 \\
-res:ialu 1 \\
-res:imult 1 \\
-res:memport 1 \\
go.ss 9 9 go.in
```

You will see on the screen the moves made by the computer playing against itself. It should end with move #59, when both players pass. On swan, it takes ∼7 minutes to complete the execution (indicated by “Game over”).

3. Look in the log file `go.log` after the line:

```bash
sim: ** simulation statistics **
```

Note: all the following numbers can be a little different, depending on your OS. The first thing to look at is “`sim_num_insn`”, it should be about 132, 968, 382. This is total number of committed instructions. The most important number is “`sim_cycle`”, i.e. the total number of cycles required to execute the code. It should be about 250, 959, 927. This means that on your new computer, with a clock period of 10ns, the program will take 2.6s. It seems that your CPU needs on average 2 cycles for each instruction. The line “`sim_CPI`” gives the exact ratio, 1.8874. The rest of the log file contains useful statistics; their name is pretty intuitive. I recomend you to look first at the `il1.miss_rate`, `dl1.miss_rate` and `ul2.miss_rate`.

**The Real Task:** You want to speed up the CPU, but you were approved an extra budget of only $100 (per CPU). Listed are the possible improvements you can make, and their price. Remember that you cannot exceed $100!

A: You can improve the il1 cache. The options available are shown below.

- 8k, 32 byte, 1 way = (256:32:1:1) $0
- 8k, 32 byte, 2 way = (128:32:2:1) $5
- 8k, 32 byte, 4 way = (64:32:4:1) $15
- 16k, 32 byte, 1 way = (512:32:1:1) $10
- 16k, 32 byte, 2 way = (256:32:2:1) $15
- 16k, 32 byte, 4 way = (128:32:4:1) $25
- 32k, 32 byte, 1 way = (1024:32:1:1) $20
- 32k, 32 byte, 2 way = (512:32:2:1) $30
- 32k, 32 byte, 4 way = (256:32:4:1) $50

B: You can also improve dl1. Options:

- 8k, 32 byte, 1 way = (256:32:1:1) $0
- 8k, 32 byte, 2 way = (128:32:2:1) $5
- 8k, 32 byte, 4 way = (64:32:4:1) $12
- 16k, 32 byte, 1 way = (512:32:1:1) $10
- 16k, 32 byte, 2 way = (256:32:2:1) $14
- 16k, 32 byte, 4 way = (128:32:4:1) $22
- 32k, 32 byte, 1 way = (1024:32:1:1) $20
• 32k, 32 byte, 2 way = (512:32:2:1) $28
• 32k, 32 byte, 4 way = (256:32:4:1) $45

Note: the specs are the same, but the prices are a little bit lower.

C: You can improve the ul2 cache. Options:

• 64k, 64 byte, 2 way = (512:64:2:1) $0
• 64k, 64 byte, 4 way = (256:64:4:1) $11
• 64k, 128 byte, 2 way = (256:128:2:1) $3
• 64k, 128 byte, 4 way = (128:128:4:1) $16
• 128k, 64 byte, 2 way = (1024:64:2:1) $10
• 128k, 64 byte, 4 way = (512:64:4:1) $21
• 128k, 128 byte, 2 way = (512:128:2:1) $14
• 128k, 128 byte, 4 way = (256:128:4:1) $27
• 256k, 128 byte, 2 way = (1024:128:2:1) $30
• 256k, 128 byte, 4 way = (512:128:4:1) $55

*How to do: replace the corresponding sim-outorder flags (i.e. the ones starting with -cache) with the format given in parenthesis after the option you choose. As a simple check, remember that cache_size = nsets * assoc * block_size.*

D: You may increase the number of integer ALUs:

• 1: $0
• 2: $8
• 3: $18
• 4: $30
• 5: $45
• 6: $60

*How to do: modify the -res:ialu flag.*

E: You may increase the number of integer MULTs:

• 1: $0
• 2: $12
• 3: $30
• 4: $55
• 5: $70
• 6: $100

*How to do: modify the res:imult flag.*

F: You may increase the number of memory ports:

• 1: $0
• 2: $5
• 3: $14
• 4: $28
• 5: $50
• 6: $75
How to do: modify the res:memport flag.

G: You may invest in faster silicon, increasing the clock frequency:

- 100 MHz: $0
- 110 MHz: $7
- 120 MHz: $18
- 130 MHz: $32
- 140 MHz: $50
- 150 MHz: $75

This improvement obviously does not alter the number of cycles taken by the program, so you do not have to modify any sim-outorder flags.

Very important: modifying any flags other than the ones specified above, and/or with different values than above, renders that particular solution invalid. The same for spending more than $100. However, feel free to do so in the process of searching for your solution.

First hint: your core can handle 4 instructions per cycle, if there are no data/control hazards. However, how many execution units do you have in the original design?

Second hint: try to obtain the best execution time for the $100 you got. Spending less than $100 is not a bonus (who cares about corporate money?).

Third hint: The above prices are reported to you by the low-level designers; they are building costs. You will find that many times a cheaper option can be more efficient than a more expensive one.

Coolness: let your execution time be \( t \). The execution time of the best solution will not be disclosed, but let us denote it by \( b \). Then, let

\[
coolness = 1.05 \times \frac{b}{t}
\]

Note that your solution may have \( coolness \geq 1 \) even if you do slightly worse than \( b \).

Do either i) or ii):

i) If you think that your \( coolness \geq 1 \), hand over only the simulation log for your chosen optimization, with no comments. That log includes by default the optimizations you have chosen. You will get maximum points if \( coolness \geq 1 \). However, if \( coolness < 1 \), you will be severely penalized. Please, no grading complaints! If you are unsure about the \( coolness \) of your solution, follow the option below.

ii) Describe in detail your experiment. Justify your comments by attaching simulation logs. I am interested in seeing the conclusions you drew after each simulation, and at each step why you chose the options that you chose. Regardless of your coolness factor, you can get near-maximum points for a well written exercise.

Final Note: I regard i) cooler than ii), but choose i) at your own risk.