**Processor Width Changes:**

Varying the processor width, as well as increasing its associated resources systematically decreased the CPI, and thus increased the speedup, compared to the baseline of the width 1, in-order processor. By increasing the processor width and its associated functional units, more instructions can be fetched, decoded, issued, and committed. the Instruction-Level-Parallelism[[1]](#footnote-1) gains achieved by increasing the width are still limited by the functional units of the processor. The greatest gains came from switching to out of order execution. Out of order execution allows the same gains as in order, but reduces the bottleneck effect of the functional units by allowing other instructions to execute. The switch to out of order execution caused an average 38% decrease in CPI between processors of the same width.

*Table A:* **Compress95 -** *Cycles per Instruction (CPI) and relative speedup over the width 1 processor when varying the width*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Width | Integer ALUs | Integer Multipliers | Memory Ports | Order | CPI | Relative Speedup  (Relative to “basic” processor with in order execution) |
| 1 | 1 | 1 | 1 | In order | 1.4873 | 1.000 |
| 2 | 2 | 1 | 1 | In order | 1.3308 | 1.1176 |
| 4 | 3 | 2 | 2 | In order | 1.2890 | 1.1538 |
| 8 | 6 | 3 | 2 | In order | 1.2829 | 1.1593 |
| 1 | 1 | 1 | 1 | Out of order | 1.4663 | 1.0143 |
| 2 | 2 | 1 | 1 | Out of order | 0.8340 | 1.7833 |
| 4 | 3 | 2 | 2 | Out of order | 0.5941 | 2.5035 |
| 8 | 6 | 3 | 2 | Out of order | 0.5426 | 2.7411 |

**Branch Prediction Strategy:**

The goal of the branch prediction strategy is to minimize the amount of branch misses, which cause work to be wasted. Because the CPI increased in both Taken to Not Taken, it can be inferred that the Compress95 program needs to take a branch more frequently than not taking the branch. This change held for both in order/out of order execution. However, there is a distinct CPI decrease observed between the dynamic 2-Level and Perfect compared to the static Taken/Not Taken strategies[[2]](#footnote-2). 2-Level and Perfect strategies “remember” the program’s recent branching, and predicts accordingly. As shown in Table B, the Perfect, out of order execution achieved a speedup of 4.1109, with no changes to the processor width or functional units.

*Table B:* **Compress95 -** *CPIs and speedups of processors with width 4, 3 ALUs, 2 IMUs, and 2 MPs*

|  |  |  |  |
| --- | --- | --- | --- |
| Branch Predictor | Order | CPI | Relative Speedup  (Relative to “branch prediction” processor with in order execution and a “taken” branch predictor) |
| Taken | In order | 1.8943 | 1.0000 |
| Not taken | In order | 1.8971 | 0.9985 |
| 2 level | In order | 1.2909 | 1.4674 |
| Perfect | In order | 1.2591 | 1.5045 |
| Taken | Out of order | 1.1694 | 1.6199 |
| Not taken | Out of order | 1.1770 | 1.6094 |
| 2 level | Out of order | 0.6459 | 2.9328 |
| Perfect | Out of order | 0.4608 | 4.1109 |

**Project Description for Lab 2**

**CSCE 692: Design Principles of Computer Architecture**

Due: 1100 22 January 2019

Assigned Benchmark: Compress95

**Project Objective:** The objective of this project is to give the student experience in (1) the simulation and analysis of microprocessor architectures at a detailed level, and (2) making design decisions and tradeoffs based on performance *and* cost.

**Project Overview**: This project will use the SimpleScalar simulation tool. For this phase of the project, each student will independently run their own simulations and write up their own report. The second phase of the project will take place later in the course. Under no circumstances should students turn in exact copies of the project. In this project, you will have to learn to use the SimpleScalar architectural simulator and you will have to run many simulations (in some cases simulation runs can take a considerable amount of time). Do not leave it until the last minute and expect to be able to finish.

**Report Format:** The hard copy report must include:

(a) 1-page executive summary highlighting any design decisions and your individual analysis of results

(b) the report itself, including tables of execution results (see examples on page 3)

(c) List of references used

(d) Appendix including relevant code/scripts used, emailed separately to save paper.

**Learning to use SimpleScalar –** This project must be done **individually**. Follow the documentation provided in *CSCE692Project.tar.gz* (if *tar –xvf* does not work, then try *gunzip* then *tar –xf* ).

SimpleScalar is really a collection of many simulators. You will use the “***sim-outorder***” simulator. This simulator is built for a *PISA* (portable instruction set architecture) platform and binaries compiled to the *PISA* target are supplied and can only be “executed” in the simulator.

**Getting the software:** You have a few options for getting SimpleScaler

1. Install on your own Linux box or Windows box running Cygwin. Download *simplesim-3v0e.tgz* from <http://www.simplescalar.com/> and follow the README to build a PISA target. You may get compiler warnings, but should install cleanly. The install script will determine if you have a *little vs. big-endian* architecture and compile appropriately. (For an x86 platform you will most likely be little… if you have a big-endian machine you will need a different compiled benchmark program). Once installed, ensure the *sim-\** executables, *pipeview.pl*, and *textprof.pl* are part of your path (perhaps copy to *~/bin*). Run the test-scripts from the README; you will get a lot of output, just ensure the “diff” portions show no file differences between your runs and baseline output.

2. AFIT/EN Linux account.(I have some dated instructions on this, no guarantee of correctness.)

For both options, go to the supplied benchmarks and follow the README to test the simulator for your assigned benchmark. This test will ensure your output matches pre-determined baselines. Finally, you may want to refresh (or teach) yourself a little (bash) shell programming and review “redirection” to help you run all your test cases.

**A. Varying the “width” of the processor**

For this part of the project we will explore increasing the number of instructions that can be issued per cycle. For each of the processor widths, the following table shows the available execution resources.

**Table 1: Execution Resources**

|  |  |  |  |
| --- | --- | --- | --- |
| **machine**  **Width** | **Integer ALUs** | **Integer Multipliers** | **Memory Ports** |
| 1 | 1 | 1 | 1 |
| 2 | 2 | 1 | 1 |
| 4 | 3 | 2 | 2 |
| 8 | 6 | 3 | 2 |

During simulation in part A, the following SimpleScalar (the “***sim-outorder***” simulator) arguments are required (details for each of these arguments along with many others can be found by referencing “documentation/QuickReference.pdf”):

* **Machine width:** -fetch:ifqsize <int> -decode:width <int> -issue:width <int> -commit:width <int>
* **# of integer ALUs:** -res:ialu < int>
* **# of integer multipliers:** -res:imult <int>
* **# of memory ports:** -res:memport <int>
* **Execution order:** -issue:inorder <true | false>

For this problem, run *your assigned benchmark* using these four configurations for **both** in-order issue and out-of-order issue. To issue instructions out-of-order, SimpleScalar uses a “Register Update Unit”. While this is different from the Tomasulo approach described in class and in the book, it’s objective is the same, and understanding how it works is not critical for the purposes of this project, especially since no real microprocessors utilize this scheme. Report the results of your 8 simulations in a table showing the CPI achieved for each configuration. Explain why the CPIs vary (if they do).

Relative to a processor of width 1 with in-order issue, what are the speedups of the various configurations? Report the speedups as part of the table showing the CPI achieved for each configuration. Once again, explain the CPIs attained.

**B. Varying the branch prediction strategy**

With a processor with **width 4** and the corresponding execution resources shown in the table above, vary the branch prediction scheme (controlled by the “-*bpred*” parameter) among the “*taken*”, “*nottaken*”, “*2lev*”, and “*perfect*” predictors. Do this for **both** in-order and out-of-order processors. Report the results of your 8 simulations in a table showing the CPI achieved for each configuration. Relative to an *in-order* processor of width 4 using a *taken* branch predictor, what are the speedups of the various configurations? Again, report the speedups in your table and explain.

**HINTS**

**There are command line examples in the example\_script\_xxx.sh file for both running sim-outorder and diff. You should look at those and figure out how they work.**

Note that the output here is the actual output of the program, so for *your assigned benchmark* your output will be the actual ouput of the program – redirected from *stdout*. As long as the program output matches the supplied baseline output (e.g., *diff*) your program is operating correctly in the PISA format, but what you are interested in is the statistical measurements from SimpleScalar which are output to screen (e.g., CPI) via the *stderr* output. Therefore to save the simulator output if scripting, you must redirect the *stderr* to a file.

**Tips:** When running SimpleScalar, it is always a good idea to check the output (using *diff* against the reference output) to make sure that (a) there were no simulator errors in the run and (b) that the configuration that was used matches the configuration that you were trying to use.

If you are not familiar with bash shell scripting (or if it has been a while) try searching for tutorials and examples online. I found this one: <http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html>

Execution table examples:

*Table A:* **Compress95 -** *Cycles per Instruction (CPI) and relative speedup over the width 1 processor when varying the width*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Width | Integer ALUs | Integer Multipliers | Memory Ports | Order | CPI | Relative Speedup  (Relative to “basic” processor with in order execution) |
| 1 | 1 | 1 | 1 | In order | 1.4873 | 1.000 |
| 2 | 2 | 1 | 1 | In order | 1.3308 | 1.1176 |
| 4 | 3 | 2 | 2 | In order | 1.2890 | 1.1538 |
| 8 | 6 | 3 | 2 | In order | 1.2829 | 1.1593 |
| 1 | 1 | 1 | 1 | Out of order | 1.4663 | 1.0143 |
| 2 | 2 | 1 | 1 | Out of order | 0.8340 | 1.7833 |
| 4 | 3 | 2 | 2 | Out of order | 0.5941 | 2.5035 |
| 8 | 6 | 3 | 2 | Out of order | 0.5426 | 2.7411 |

*Table B:* **Compress95 -** *CPIs and speedups of processors with width 4, 3 ALUs, 2 IMUs, and 2 MPs*

|  |  |  |  |
| --- | --- | --- | --- |
| Branch Predictor | Order | CPI | Relative Speedup  (Relative to “branch prediction” processor with in order execution and a “taken” branch predictor) |
| Taken | In order | 1.8943 | 1.0000 |
| Not taken | In order | 1.8971 | 0.9985 |
| 2 level | In order | 1.2909 | 1.4674 |
| Perfect | In order | 1.2591 | 1.5045 |
| Taken | Out of order | 1.1694 | 1.6199 |
| Not taken | Out of order | 1.1770 | 1.6094 |
| 2 level | Out of order | 0.6459 | 2.9328 |
| Perfect | Out of order | 0.4608 | 4.1109 |

1. Appendix C Slides – Pipelining, Dr. Scott Graham [↑](#footnote-ref-1)
2. *Computer Architecture: A Quantitative Approach,* Appendix C [↑](#footnote-ref-2)