

Figure 1: Changes to sim-scalar-cpen411.c

Two extra if statements were added to the decode stage’s stall and a change to the sim\_cycle comparison. “Sim\_cycle + 1” allowed for immediate reading from the WriteBack stage. The next check for the opcode of the source register, src[i], to see if it is a Load instruction. Load instruction are the only cause of a stall that isn’t a branch, which is handled already by the code provided. If the source register is a Load but has gotten to the WriteBack stage it will not reach the final nested if, however if the source register is a load and only just finished the execute stage then the program will stall in decode.

Assuming the geometric mean is the speedup values multiplied by each other and then put to the power of 1/n where n is the number of speedups used the value found is as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | gcc | fpppp | vpr | go |  | geometric mean |
| speedup | 1.3606 | 1.2673 | 1.4795 | 1.6155 |  | 1.4248 |

From these values it can be seen that pipelining has had a significant increase in speed. The most largest change can be seen in “go”. Most likely this is due to a large amount of instructions that end in the execute stage or a small amount of dependencies. The smallest changes were in “gcc” and “fpppp” which might be due to a large amount of dependencies on loads or a large amount of branches. From the first lab it is found that about 39% of instructions were loads for “fpppp” which can be the cause of less of a speed up. “go” had 20% loads and 20% branches, so it can be assumed from these results not many of the loads were loading data that was depended on in the next two instructions.

**Results from part B)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | gcc | fpppp | vpr | go |
| CPI\_ii | 1.3052 | 1.4198 | 1.2825 | 1.2897 |
| CPI\_b | 1.3473588 | 1.2918557 | 1.3035187 | 1.313628 |
| Speedup\_b | 1.317986 | 1.392802617 | 1.4555986 | 1.586065 |

**\***fppp was more than 3% off

To calculate the CPI of part B I understood that a stall between instructions only happen at branches or at a Memory instruction that needs to be used in an ALU operation. I assumed that each of these stalls would only stall by one cycle, and found the frequency of each to come out with the given CPI\_b in the above table. All the values found are shown below.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Stalled-Loads | num\_cond | num\_uncond | num\_inst |  | freq\_load | freq\_cond | freq\_uncond | CPI |
| gcc | 29489117 | 30432486 | 9550147 | 200000000 |  | 0.147445585 | 0.15216243 | 0.047750735 | 1.347359 |
| fpppp | 41693701 | 1571413 | 532920 | 150067427 |  | 0.277833117 | 0.01047138 | 0.003551204 | 1.291856 |
| vpr | 31174740 | 21754283 | 7774719 | 200000000 |  | 0.1558737 | 0.108771415 | 0.038873595 | 1.303519 |
| go | 21462864 | 15995579 | 4243183 | 132965196 |  | 0.161417158 | 0.120298992 | 0.031911982 | 1.313628 |

To find the CPI:

(Stall-Load+num\_cond+num\_uncond)/num\_inst + 1 = CPI