Anthony Alayo 997487401
Freddy Chen 997363124

ECE552 Lab 1 – Performance Measurements

% Performance Drop versus an ideal pipeline with CPI of 1.0, based on a run of gcc.eio

Question 1

CPI = 1.6425 --> 64.25% performance drop

CPI = ((sim_num_insn + 5 - 1) + single_cycle_stalls_q1 + (double_cycle_stalls_q1 * 2)) / (sim_num_insn + 5 - 1)

This equation is derived by taking the total number of instructions and calculating how many cycles a 5 stage pipeline needs to complete all the instructions, assuming that each stage takes one cycle. We are effectively adding 4 to the total number of instructions because it takes 4 extra cycles for the pipeline to fill up with instructions. Single and double cycle stall penalties are added to the total and a ratio is taken to figure out the performance drop from the ideal CPI of 1.0.

Question 2

CPI = 1.3887 --> 38.87% performance drop

CPI = ((sim_num_insn + 6 - 1) + single_cycle_stalls_q2 + (double_cycle_stalls_q2 * 2)) / (sim_num_insn + 6 - 1)

The form of this equation is similar to question 1, with the exception of a 6 stage pipeline, meaning we effectively add 5 to the total number of instructions because it takes 5 extra cycles for the pipeline to fill up with instructions.

Question 3

CPI = 1.2028 --> 20.28% performance drop

CPI = (((sim_num_insn + 6 - 1) * (sim_num_refs / sim_num_insn) + (sim_num_insn + 4 - 1) * (1 - (sim_num_refs / sim_num_insn))) + single_cycle_stalls_q3 + (double_cycle_stalls_q3 * 2)) / ((sim_num_insn + 6 - 1) * (sim_num_refs / sim_num_insn) + (sim_num_insn + 4 - 1) * (1 - (sim_num_refs / sim_num_insn)))

Again, the form of the equation is similar to questions 1 and 2. However, the total number of cycles needed to complete all instructions is now calculated by summing the total number of cycles needed to complete memory related operations (6 cycles per instruction) and the total number of cycles needed to complete non-memory related operations (4 cycles per instruction).

Anthony Alayo 997487401
Freddy Chen 997363124

Question 1 Micro-benchmark Explained

For the first question in the problem statement, we had to look at the effect of RAW hazards in a 5 stage pipeline with no bypassing/forwarding and control hazards ignored. This scenario brought upon only 2 hazard cases:

next inst dependence -> RAW Hazard

	c1	c2	c3	c4	c5	c6	c7	c8
add rs1, rs2 -> **r1**	F	D	X	M	W			
sub **r1**, rs2 -> rt		F	d*	d*	D	X	M	W

2 inst later dependence -> RAW Hazard

	c1	c2	c3	c4	c5	c6	c7	c8
add rs1, rs2 -> **r1**	F	D	X	M	W			
add (independent)		F	D	X	M	W		
sub **r1**, rs2 -> rt			F	d*	D	X	M	W

Looking at this, we needed to test for 2 cases in our micro-benchmark. One scenario is when there is a dependent instruction immediately after an instruction, and the other scenario is when there is a dependent instruction 2 instructions down the pipeline. Each of these scenarios result in different stall amounts, as displayed above in the table.

I emulated both these scenarios in my micro-benchmark. Looking at the code, I have...

```
asm("nop");

for(i = 1; i <= max; i++) {
   // the below code creates our
   // double cycle stall
   temp = temp + 3;
   sum = temp + 3;

   // the below code creates our
   // single cycle stall
   temp2 = temp2 + 5;
   dummy = dummy + 1;
   sum2 = temp2 + 5;
}
```

```
  asm("nop");
```

I placed no ops around my loop code so that I could identify it in assembly, regardless of the optimizations done. In the code you can see that I commented an area for a double cycle stall, and an area for a single cycle stall. I did this by using simple dependent instructions. I compiled my code using the –O2 optimization level and resulted with this assembly:

```
   #APP
     nop
   #NO_APP
     li  $3,0x00000001      # 1
     blez    $4,$L16
  $L18:
     addu    $17,$17,3    temp = temp + 3
     addu    $20,$17,3    sum = temp + 3           <-- DATA HAZARD (OURS, DOUBLE CYCLE)
     addu    $18,$18,5    temp2 temp2 + 5
     addu    $19,$19,1    dummy = dummy + 1
     addu    $21,$18,5    sum2 = temp2 + 5          <-- DATA HAZARD (OURS, SINGLE CYCLE)
     addu    $3,$3,1
     slt $2,$4,$3        bit for branch decision     <-- DATA HAZARD (INHERENT, DOUBLE CYCLE)
     beq $2,$0,$L18      check bit, loop or not loop <-- DATA HAZARD (INHERENT, DOUBLE
CYCLE)
  $L16:
   #APP
     nop
```

Notice here that my data hazards are clearly present, as well as 2 inherent data hazards due to the handling of the loop by the compiler. In total, that is 1 single cycle stall and 3 double cycle stalls per loop iteration.

I have the number of loops being passable with the execution of my code. Thus, if you pass N as the number of loops, you should get approximately N * 4 RAW hazards for question 1 (assuming N is sufficiently large to mitigate other hazards existing in overhead code).

Running this micro-benchmark using N = 100000, I get the results:

```
sim_num_RAW_hazard_q1       406242 # total number of RAW hazards (q1)
single_cycle_stalls_q1      101547 # total number of single cycle stalls for q1
double_cycle_stalls_q1      304695 # total number of double cycle stalls for q1
```

These micro-benchmark collected statistics validate the correctness of my code.