Max Williams

Assignment 1

CSE 611
September 1, 2020

Problem 1

Architecture or *instruction set architecture* refers to the architecture as seen by the programmer or compiler. This consists of the registers, instructions and their operands, and other details as they directly affect computation. The architecture is a logical abstraction of the *microarchitecture* which includes details which affect performance but (ideally) do not affect the result of the computation.

The line between these two can be blurred if the programmer is aware of the microarchitecture and attempts to write code which causes microarchitecture level optimization, such as structuring code so that the branch predictor works more accurately or read-after-writes can be done in zero steps by reusing data. The line can be pathologically blurred in the other direction if bugs in the microarchitecture cause a divergence between the expected behavior as per the architecture and the actual behavior of the system, such as using branch prediction and cache latency to take data from illegal location as in the Specter and Meltdown exploits.

Problem 2

$$\begin{aligned}0.15 \text{ stores} \times 1\frac{\text{cycles}}{\text{store}} + 0.25 \text{ loads} \times 2\frac{\text{cycles}}{\text{load}} \\ + 0.2 \text{ branches} \times 4\frac{\text{cycles}}{\text{branch}} + 0.4 \text{ integer ALUs} \times 1\frac{\text{cycles}}{\text{integer ALU}} \\ = 1.85\frac{\text{cycles}}{\text{instruction}}\end{aligned}$$

Problem 3

```
for (i=1;i<100;i++)
{
    y[i] = x[i]/c;    // S1
    x[i] = x[i] + c;  // S2
    z[i] = y[i] - c;  // S3
    y[i] = c - y[i];  // S4
}
```

S3 and S1 create a RAW dependency (true dependency) on y.
S1 and S4 create a WAW dependency (output dependency) on y.
S1 and S2 create a WAR dependency (anti-dependency) on x.
If S2 is not an atomic operation, then it creates a RAW hazard (true dependency) on x. The same is true of S4 for y. However, on a CISC architecture these can possibly be done with a single instruction using indirection and there would not be these hazards.

The looping (and branch at the end of the loop) are independent of the data, and each iteration is independent, so there is no concern over other dependencies created between the loop iterations. Hazards related to the loop variable 'i'.

Problem 4

```
(a) ADD R1, R1, #4
    LD  R2, 7(R1)
(b) ADD R3, R1, R2
    ST  R2, 7(R1)
(c) ST  R2, 7(R1)
    ST  R3, 200(R7)
(d) BEZ R1, place
    ST  R1, 7(R1)
```

Problem 4.1

Code segment *(a)* has a true dependency, read after write. `R1` is written to upon completion of the ADD instruction, and the `LD` instruction depends on `R1` for offsetting the address it loads from. Once `R1` is written to, the `LD` instruction is free to run without this hazard.

Code segment *(b)* has no dependencies. The first instruction reads `R1` and `R2` and writes `R3`. The second reads `R1` and `R2` and writes to some location in memory. Thus only "read after read" takes place and there are no data hazards.

Code segment *(c)* has a potential output dependency. If `7(R1)` and `200(R7)` happen to point to the same location in memory, then that location is written to by the first instruction then the second. This is resolved once the first instruction has finished writing, then the second is free to (possibly) overwrite it.

Code segment *(d)* creates a branch hazard, and the `ST` instruction cannot write to memory until it is known whether is `BEZ` instruction is going to branch. This is because whether or not the `ST` instruction is ran depends on the result of `BEZ`. Thus the `ST` must be delayed until after the branch is finished (possibly) writing to the instruction pointer.

Problem 4.2

Code segment *(a)* does not allow for out of order execution because the second line depends on the first. Code segment *(b)* can be reordered without issue. Code segment *(c)* can only be reordered if `7(R1)` and `200(R7)` happen to point to different locations. Code segment *(d)* cannot be reordered because it is unknown whether the `ST` instruction will be ran at all. Branch prediction can be used, for instance by caching the address the `ST` instruction is going to use if it is expected to run.