Computer Architecture (Sheet #3)

Marius Gavrilescu

1. Sequential has latency 320ps, throughput 3.13 GIPS.
   Pipelined has latency 360ps, throughput 8.33 GIPS.

2. (a) `addq` reads the register written to by `irmovq`. Additionally, `mrmovq` reads the register written to by `addq`. These two data hazards will be addressed by inserting 3 bubbles between each pair of consecutive instructions.

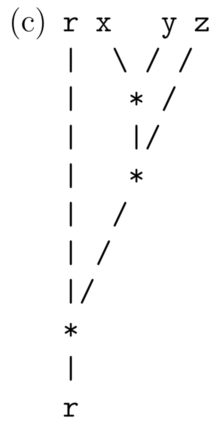
   (b) When the `jnz loop` instruction is executed and it does not branch, the processor will replace the two instructions in the first two stages of the pipeline with bubbles. These instructions are the `addq` and `subq`.

   When we get to the `ret` instruction, the processor will stall for 3 cycles (insert three bubbles after `ret`), because it cannot predict the branch at all.

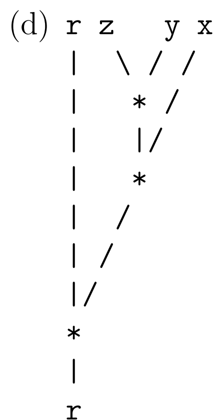
   (c) When we get to the `halt` instruction, the state of every pipeline stage it passes through becomes HLT. Any further instructions are dropped (a bubble is inserted instead of the next instruction).

3. (a)
```
r x y z
|/ / /
* / /
|/ /
* /
|/
*
|
r
```
   No parallelism possible, we get 5 CPE.

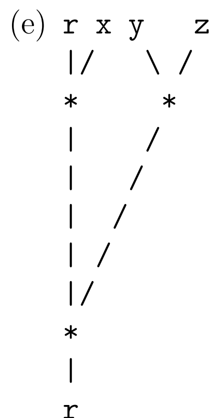
   (b)
```
r x   y z
|  \ / /
|   * /
|  / /
| / /
|/ /
* /
|/
*
|
r
```
   The multiplication x * y for iteration (i+1) can be done while we are doing iteration i, so we only have two multiplications on the critical path. This gives us 3.33 CPE.

(c) 

With sufficient floating point units we can compute x * y for iteration (i+2) at the same time as (x * y) * z for iteration (i+1) and r * ((x * y) * z) for iteration i. This gives us 1.66 CPE.

(d) 

This is identical to case (c).

(e) 

First two multiplications can be done in parallel, third depends on the two results. This gives us 3.33 CPE.

4. Surely $d = r \cdot c$, so $b_r + b_c$ is constant whatever our choice of r and c. If instead we are minimising $\max(b_r, b_c)$, we can try to make our array as square as possible, i.e. $r = c$ or $r = 2 \cdot c$.

   This gives us $r = c = 2^2$ for the first two cases, $r = 2^4, c = 2^3$ for (c), $r = 2^5, c = 2^4$ for (d) and $r = c = 2^5$ for (e).

5. The array is stored in memory as a sequence of `struct points`. Each point is stored as a sequence of three `floats`, the first three being the `vel` array, and the last three being the `acc` array. Hence (c) is the worst option, and the access patterns of the other options are:
   1, 4, 2, 5, 3, 6 for (a)
   4, 5, 6, 1, 2, 3 for (b)
   so (b) has better spatial locality than (a).

   It would be better to access the memory sequentially. The best solution for this problem is to exploit the fact that the float value 0 is represented as all zeroes in memory, which allows us to use `memset`:

```
#include <string.h>
void clear(point *p, int n) {
  memset(p, 0, sizeof(point) * n);
}
```

6.

Cache	m	C	B	E	S	t	s	b
1	32	1024	4	1	256	12	8	2
2	32	1024	8	4	32	24	5	3
3	32	1024	32	32	1	26	1	5

7. The 13 bits are `TTTTTTTTSSSBB` where `TTTTTTTT` is the tag, `SSS` is the set index, and `BB` is the block offset.

   The addresses that hit in set 5 are $0xE20 + 5 * 4 + b = 0xE34 + b$, so 0xE34, 0xE35, 0xE36, 0xE37.

   0x0D53 has block offset 3, set 4, tag 0x6A. It misses.
   0x0CB4 has block offset 0, set 5, tag 0x65. It misses.
   0x064D has block offset 1, set 3, tag 0x32. It hits, value 8.

8. On first iteration: we load `a` into the cache (set 0), then we load `b` into the cache (set 0), evicting `a`.
   On next iteration: we load `a` into the cache (set 0), then we load `b` into the cache (set 0), evicting `a`.

   So our cache miss rate is 100%.

   To fix this we can switch to a 2-way cache. This way on first iteration we'll load both `a` and `b` in the cache (set 0), and on next 3 iterations we'll get cache hits. This gives us a 25% cache miss rate.