Homework 10

1. PIPELINE EXCEPTION HANDLING

   You are required to extend the PIPE implementation so that it will reboot after any exception. The CPU should only reboot when the exception reaches the W stage. After the reboot PC is reset back to 0 and other registers, CC and memory should be preserved. For simplicity, you *DO NOT* need to consider the combination between reboot hazard and other hazards.

   (a) What is the `f_pc` and `W_stall` now?

   ```
   f_pc = [
       W_stat in { SADR, SINS, SHLT }: 0 # Handle reboot
       # Original codes
       ...
   ];

   W_stall = 0;
   ```

   (b) How many other instructions have entered the pipeline when the first instruction after reboot is in the FETCH stage?

   3. (instructions in the DECODE, EXECUTE and MEMORY stage)

   (c) How do you prevent these instructions (instructions that *AFTER* the exception and *BEFORE* the first instruction after reboot) from modifying registers, CC and memory?

   Suppose the current state of the pipeline is:

   ```
   | Stage | Instruction                                    |
   |-------+------------------------------------------------|
   | F     | intrR: The first Instruction after reboot      |
   | D     | intrC                                          |
   | E     | intrB                                          |
   | M     | intrA                                          |
   | W     | intrE: The instruction that causes exception   |
   ```

   In the *original* exception handling of the pipeline, when intrE is in the W stage, intrA has already been bubbled (think of why), and intrB is going to be bubbled in the next cycle. And intrC will be bubbled when it enters the M stage because intrE is stalled in the W stage in the original design.

   However now, our CPU is no longer stuck when an exception reaches the W stage. Since intrE is not stalled in the W stage which keeps the signal `W_stat` high, intrC will successfully go through the W stage and may modify registers.

   To solve this problem, you can signal a bubble to the E stage which will clear the intrC in the next cycle:

```
E_bubble =
    W_stat in { SADR, SINS, SHLT } ||
    ... # Original codes
```

2. MACHINE INDEPENDENT OPTIMIZATION

Suppose we have some codes as below.

```
typedef struct {
        int vals[3];
} block_t;

typedef struct {
        int length;
        block_t *blocks;
} blocklist;

int get_length(blocklist *bl)
{
        return bl->length;
}

block_t* get_blocks(blocklist *bl)
{
        return bl->blocks;
}

void SUM(blocklist *bl, long *dest)
{
        for (int i = 0; i < get_length(bl); i++) {
                int size = 1;
                for (int j = 0; j < 3; j++)
                        size = size * get_blocks(bl)[i].vals[j];
                *dest = *dest + size;
    }
}
```

Try to optimize the function `SUM` with a combination of optimizations you have learned in the ICS class. Comment briefly on your optimizations.

```
void SUM(blocklist *bl, long *dest)
{
        int length = get_length(bl); // reduce loop overhead
        block_t* blocks = get_blocks(bl); //reduce function call
        int tmp = 0;
        for (int i = 0; i < get_length(bl); i++) {
                int* vals = blocks[i].vals; // reduce repeated calculation
                tmp = tmp + vals[0]*vals[1]*vals[2]; //loop unrolling
        }
        *dest = tmp; //reduce memory access
}
```