Group 4 - Assignment 3

1.Structure
I_cache[512] // this is the I-cache array
D_cache[2048] // D-cache array
In cache_block struct { valid_bit, tag, uint8_t i_data[16], uint8_t d_data[8]} // cache block

Functions for cache_block struct:

```
    * cache block:
     * creat_entry(): Create a Cache_block with valid bit, tag and data
     * update_block(): Update a Cache_block with new valid bit, tag, and data.
     * get_cache_validBit():
     *              input: cache block entry
     *              output: valid bit of this entry
     * get_cache_tag():
     *              input: cache block entry
     *              output: tag of this entry
     * get_cache_data():
     *              input: cache block entry
     *              output: data of this entry
     *
     * calcu_cache_index:
     *     input: pc or address
     *     output: calculate index of this pc or address
     *
     * calcu_cache_tag:
     *     input: pc or address
     *     output: calculate tag of this pc or address
     *
     * calcu_cache_offset:
     *     input: pc or address
     *     output: calculate offset of this pc or address
```

When miss, we need to add a cache block to corresponding entry.
Create cache_block struct and put it into cache[index].

For example.

```
typedef struct Cache_block{
    int valid;
    int tag;
    void* i_data; // for I-cache
    void* d_data; // for D-cache
}* Cache_entry;
```

```
Cache_entry create_cache_entry(int valid, int tag, void* i_data, void* d_data){
        Cache_entry entry = malloc(1*sizeof(struct Cache_block));

        entry -> valid = valid;
        entry -> tag = tag;
        entry -> i_data = i_data;
        entry -> d_data = d_data;
        return entry;
}
```

Code above are cache structure and create cache function.
We add cache block use only one line code.

```
         i_cache[index] = create_cache_entry(1, tag, instr, instr);
```

May store cache block struct into i_cache[] and d_cache[]. The index of array is the index of cache.

2.Where to use cache

Remember, register access and memory access are limited:

1. Fetch: memory access read one time (i-cache)
2. Decode: register access read one time
3. Execute: no operation related to register and memory access
4. Memory: memory access read or write one time (d-cache)
5. Writeback: register access write one time

3.Address calculation

I-cache bits: [31:13][12:4][3:0] .

    Tag: pc>>13
    Index: (pc>>4) & 0x1ff
    Offset: pc & 0xf   (store 4 instruction in one block, use 16 byte for 4 instruction)

D-cache bits:[31:14][13:3][2:0]

    Tag: pc>>14
    Index: (pc>>2) & 0x7ff
    Offset: pc & 0x3   (block size of d-cache is 8byte)

We use tag and valid bit to ensure that the block we get by index is the block we want.
Offset is used for us to get every byte from the data.

4.stalls in assignment 3.
-r <num> use for control the delay circle.
READ:

I-cache and D-cache both have read access.
stall if no requested data in cache, vice versa.
The stalls for cache is happened between cache and DRAM(lower level). In other words memory_read. Now, the memory access has delay now. We can use memory_status to check if the memory access is ready for us to get the correct value.

I-cache
Spatial locality for I-cache:
Because 16 byte store, we fetch 4 instruction at once and pass it to corresponding entry of cache. The fetching of first one of these 4 instruction need to memory_read and stall. After we update the 16 byte instructions (4 instructions) in the I-cache, we can read it immediately. The next three fetching will find correct instruction and needn't to stall at all.

The instrs following have distinct program counter in sequential order. In other words, they are different instr with different pc without branch happening.

addi...	F	F	D	X	M	W				
add...			F	D	X	M	W			
sub...				F	D	X	M	W		
add..					F	D	X	M	W	
add..						F	F	D	X	M	W

D-cahce
This is situation that read on D-cache when stall happening. This means a miss in the cache. Just update cache and stall, then read. When hit in the cache, no need to store.

lb...	F	F	D	X	M	M	W

When hit in the cache, no need to store.

lb...	F	F	D	X	M	W	

WRITE:

Only D-cache has write access. In assignment 3, we use write around.
We directly write the data to memory if we miss in the cache.
We update cache and write to memory when we hit in the cache.
If we miss, nothing will exist in the cache block. So when we read this block next time, we still need the read stall.

Group 4 - Assignment 3

Just write around

sb...	F	F	D	X	M	W	

Next read need to stall because miss in this block (sb and lb operate in the same address)

sb...	F	F	D	X	M	W		
lb...			F	D	X	M	M	W

If we hit, we will update this block. When we want to read the data from this block, we must find the correct data, because we have already update it.

Next read don't need to stall because hit, and the data is correct because we update it when we do write cache operation(update).

sb...	F	F	D	X	M	W		
lb...			F	D	X	M	W	

Therefore, although memory_write have delay, we do not need to consider it, and just treat it as no delay.