Virtual address: 32 bits
Physical address: 32 bits

Page table:

Page size: 4KiB
Page offset = 12 bits
Virtual page number = 20 bits
Physical page number = 20 bits

Virtual address bits or physical address bits.

VPN or PPN	Page offset
[31:12]	[11:0]
20 bits	12 bits

Virtual address bits for page table

Virtual page number		Page offset
Base page table index	Level 2 page table index	Page offset
[31:22]	[21:12]	[11:0]
10 bits	10 bits	12 bits

To implement page table, I create a struct called PTE.

```
typedef struct page_table_entry{
        int valid;
        int* ppn; // the pointer to next level page table
}* PTE;

PTE create_PTE(int valid, int* ppn){
        PTE entry = malloc(1*sizeof(struct page_table_entry));

        entry -> valid = valid;
        entry -> ppn = ppn;
        return entry;
}

int* get_next_page_table(PTE curr_PTE){
        return curr_PTE -> ppn;
}

int calcu_base_index(uint64_t vpn){
        return vpn >> 22;
}
```

```
int calcu_second_index(uint64_t vpn){
        return (vpn >> 12) & 0x3ff;
}

int calcu_page_offset(uint64_t vpn){
        return vpn & 0xfff;
}

int physical_address(uint64_t ppn, uint64_t page_offset){
        return (ppn<<12) + page_offset;
}
```

To get physical address:

1. After using get_ptbr(), we get a pointer to base page table
2. In each pointer from get_ptbr() ~ (get_ptbr() + 1024) create a PTE, so there will be 1024 PTE structs in base page table. PTEs in first level have default valid as 0.
3. When I get a virtual address, for example, pc, I will let pc = pc >> 22 to get index of base page table, then by using get_next_page_table(get_ptbr() + pc), I get the level 2 page table pointer, and set valid bit of this PTE as 1.
4. In each of the level 2 page table I needed, I create 1024 PTEs, for each level 2 table.
5. Let pc = (pc >> 12) & 0x3ff, then I can get index of actual physical frame number by get_next_page_table function get the value from the pointer and let (ppn <<12) +page offset.

Direct-mapped TLB:

entry: 8 entries
Index bits = 3 bits

Virtual address bits for TLB

Virtual page number		Page offset
Tag	Index	Page offset
[31:15]	[14:12]	[11:0]
17 bits	3 bits	12 bits

To implement TLB, I create a struct called TLB_entry. And two array to store enties.

```
uint64_t i_TLB[8];
uint64_t d_TLB[8];
```

```
typedef struct TLB_block{
	int valid;
	int tag; // the tag from virtual page number
	int* ppn; // the physical page number
}* TLB_entry;

TLB_entry create_TLB_entry(int valid, int tag, int* ppn){
	TLB_entry entry = malloc(1*sizeof(struct TLB_block));

	entry -> valid = valid;
	entry -> tag = tag;
	entry -> ppn = ppn;
	return entry;
}

int get_TLB_valid(TLB_entry e){
	return e -> valid;
}

int get_TLB_tag(TLB_entry e){
	return e -> tag;
}

int *get_TLB_data(TLB_entry e){
	return e -> ppn;
}

// cache bits calculation
int calcu_TLB_index(uint64_t vpn){
	return vpn & 0x3;
}

int calcu_TLB_tag(uint64_t vpn){
	return (vpn>>3);
}

int calcu_TLB_offset(uint64_t vpn){
	return vpn & 0xfff;
}
```

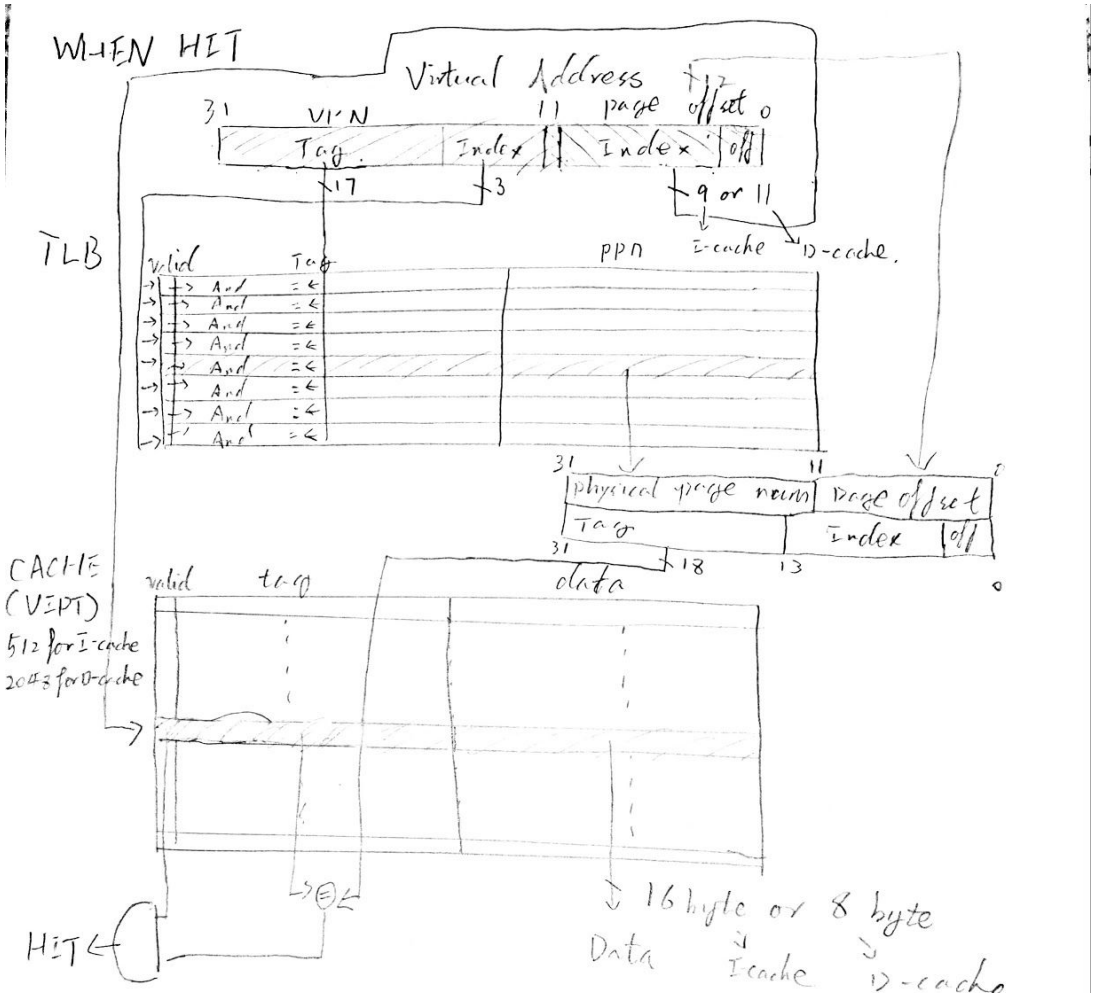
The implementation to TLB is similar to cache:

1. The data store in TLB is not the actually data, they are physical page number
2. If TLB miss, we need to walk through all level page tables (memory access).

3. If TLB hit, we get the physical page number from TLB.

VIPT(virtual address physical tag)

In VIPT cache, we need to keep [13:11] bits same in virtual address and physical address.



When VIPT cache hit, thing happens just as the picture above.

If miss, do the page walk and get the physical address, update the date and then look up the cache again.

Stall for TLB

When miss in TLB, we should do the page walk from all level page tables, and doing these will cause memory access, which leads to a stall.

After memory_read, when the memory_status becomes true, we memory_read again and get the value we want. This process will happen two times: in top-level page table and second-level page table.