Parallel Systems Architecture
Assignment 1

Saidgani Musaev
11992077

January 16, 2019

1 Introduction

In current paper we discuss solution for the first assignment of the "Parallel Systems Architecture" course. The main goal of the first assignment is building First-Level Cache for one-core CPU in SystemC "lanugage". We first discuss organization(architecture) of cache which we build here, then we discuss memory mapping and cache associativity. All sections are supported with diagrams and code snippets.

2 Architecture and Cache associativity

In current assignment we implement First-Level Cache based on Write-back strategy. At current time we didn't implement pipeline of CPU, we also have very primitive memory controller.
Figure 2 demonstrates how we connected CPU, Cache and Main Memory modules. As you can see, CPU doesnt have direct wires to the main memory. Since we don't implement Write-around caching policy, we don't need direct wires between CPU and Main Memory.

Figure 1

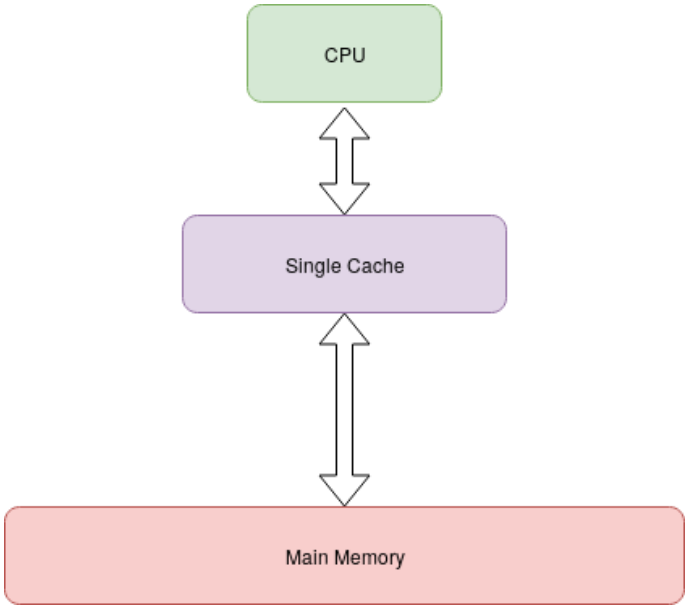

Figure 2

We don't have virtual address space, hence we don't have TLB buffer, so we don't have to make virtual-indexing and physical-tagging of cachelines. Since we don't need to translate virtual addreses to phisycal addresses, we just use given addresses without any translations and modifications. In current work we assume that we have 32-bit machine with 4KB paging. Usually, in real systems - virtual-indexing and physical tagging are done, that's why it's better to use bits in range [0-11] to encode cache set index, but as we mentioned before we don't run into this problems, because of we have direct mapping(virtually indexed and virtually tagged, e.g. GPU core). According to the assignment description, cachelines have to be 32-bytes long. Figure 1 demonstrates which bits are used for which purpose.
Bits [0-4] - offset of byte within current cacheline;
Bits [7-5] - index which points to cache set;
Bits[31-8] - these bits are used for tagging cacheline;
In the following code snippets you can see how index is being extracted from an address.

```
int addr_to_index(int addr){
        return ((addr & CACHEINDEX_MASK) >> CACHEINDEX_SHIFT) * CACHE_SET_SIZE;
}
```

As you can see, two neighbor cachelines will get to different cache sets. Its not a problem from architectural perspective, since caches are based on SRAM, so physically contiguous mapping is not required there.
According to our current implementation, CPU "doesnt know" about Main Memory, it fetches all data from the cache. When data is missed in Cache, cache itself request missed data from the main memory and fill new cacheline. As it's stated in the assignment description - we use LRU replacement policy. To implement LRU we define special cacheline structure:

```
struct cache_record_t{
        int counter;
```

```
    int state:4;
    int tag: 24;
    int data[8];
}__attribute__((packed));
```

counter - this field is needed to implement LRU, when cacheline is brough from the main memory - this counter is being set to some big value, every time when we have access to some address - we decrement counter on every cacheline from this set, except one cacheline which got hit(if such cacheline exist) and for that cacheline we actually increment counter. Of course we avoid cases of underflow and overflow the counter;
state - only keeps Valid or Invalid states for now;
tag - this field is needed to store tag of cacheline;
data - field which stores actual data;

To send full cacheline via 32bit bidirectional channel we synchronize receiver and sender modules via clock and transfer 32bit integer 8 times. We performed tests to check that modules are synchronized correctly(read value from channels on positive signal only) to not miss any chunk of data, so somewhere extra *wait*() statements are presented in the code;

3 Experiments and Conclusion

In this section we present results of simulation and share some conclusions with you.

type	CPU	Reads	RHit	RMiss	Writes	WHit	WMiss	Hitrate
dbg	0	100	45	55	0	0	0	45.00000000
fft	0	129493	36080	93413	0	0	0	27.862510
rnd	0	65536	20865	44671	0	0	0	31.837463

As we can see from the table, current LRU implementation is not that good. In current paper we didn't analyze access pattern of tracefiles. We could recommend to do it as a future work in order to improve Hitrate.