EE 6313 Advanced Microprocessor Systems
FALL – 2022

To determine the best architecture for a cache controller

BY

Madhu Narayanaswamy (1002013188)
Nithin Kashiyap Takmul Purushothamaraju (1001857031)

OVERVIEW

The goal of this project is to determine the best architecture for a cache controller that interfaces to a 32-bit microprocessor with a 32-bit data bus. While the microprocessor is general purpose in design and performs several functions, it is desired to speed up certain signal processing functions, such as a fast Fourier transform (FFT) routine.

INTRODUCTION

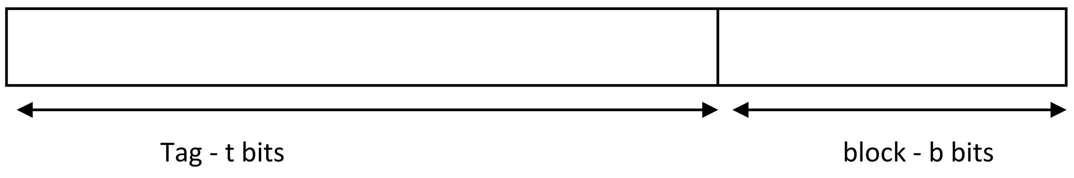
A cache is a small block of data that is designed to help the processor, retrieve data that is frequently used. It stores the data that is used by the processor to execute the instructions. The cache is placed between main memory and processor thus providing faster access time to the processor than traditional time was there between main memory and processor.

Core have access for two independent cache i.e., L1 instruction and L1 data. When the core has its data running, we go to find a way and make sure most of read and write occurs in local die. Every other core duplicate this. If we cannot find memory, we will try read in L1 instruction and data. We will go to L2 which has both instruction and data in it. It is going to be bigger memory. This is present inside CPU package. For instance, we can have multiple core processors. We can also have multiple CPU environment.

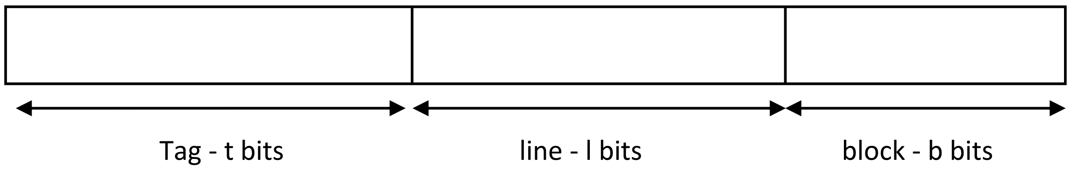
Outside the computer core we can communicate with L3 cache which could infact talk to memory. Each core has high percentage of reads and writes. All the information in L1 instruction is present in L2 cache is present in L3 cache and in memory. Memory is huge area, L3 is small, L2 is smaller. Core is able to do all the data instruction read and write from L1 by some reasons if it cannot get information it looks in L2 cache.

TYPES OF CACHE

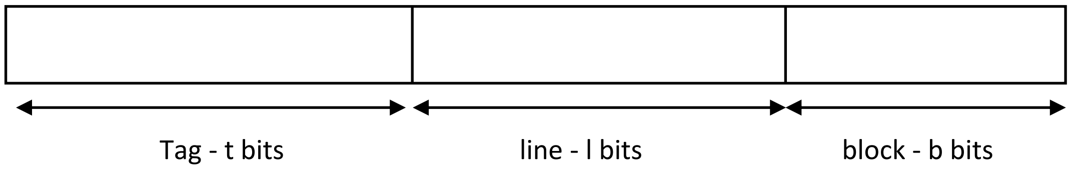
FULLY ASSOCIATIVE CACHE: A fully associative cache permits data to be stored in any cache block, instead of forcing each memory address into one block. When data is fetched from memory, it can be placed in any unused block of the cache.



DIRECT MAPPED CACHE: n a direct-mapped cache, the cache is organized into multiple sets with a single cache line per set. Based on the address of the memory block, it can only occupy a single cache line.



SET ASSOCIATIVE CACHE: An N-way set associative cache reduces these conflicts by providing N blocks in each set where data mapping to that set might be found. Each memory address still maps to a specific set, but it can map to any one of the N blocks in the set. We have used this in our project.



Here S is the size of the cache and S = L*N*B bytes and B = DW*BL

Where L - no of lines

N- no of ways

B – block in bytes

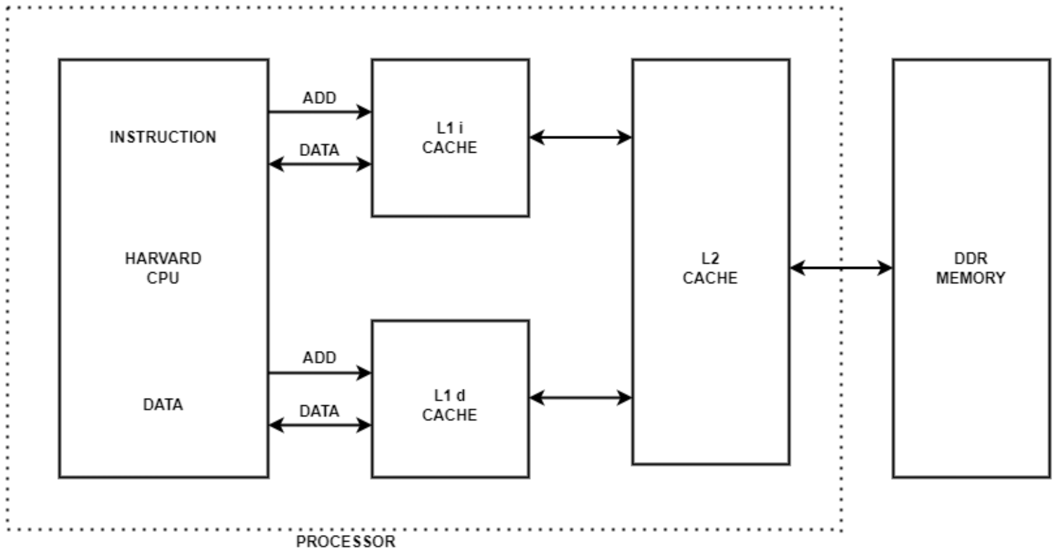
DW - data width

BL - burst length

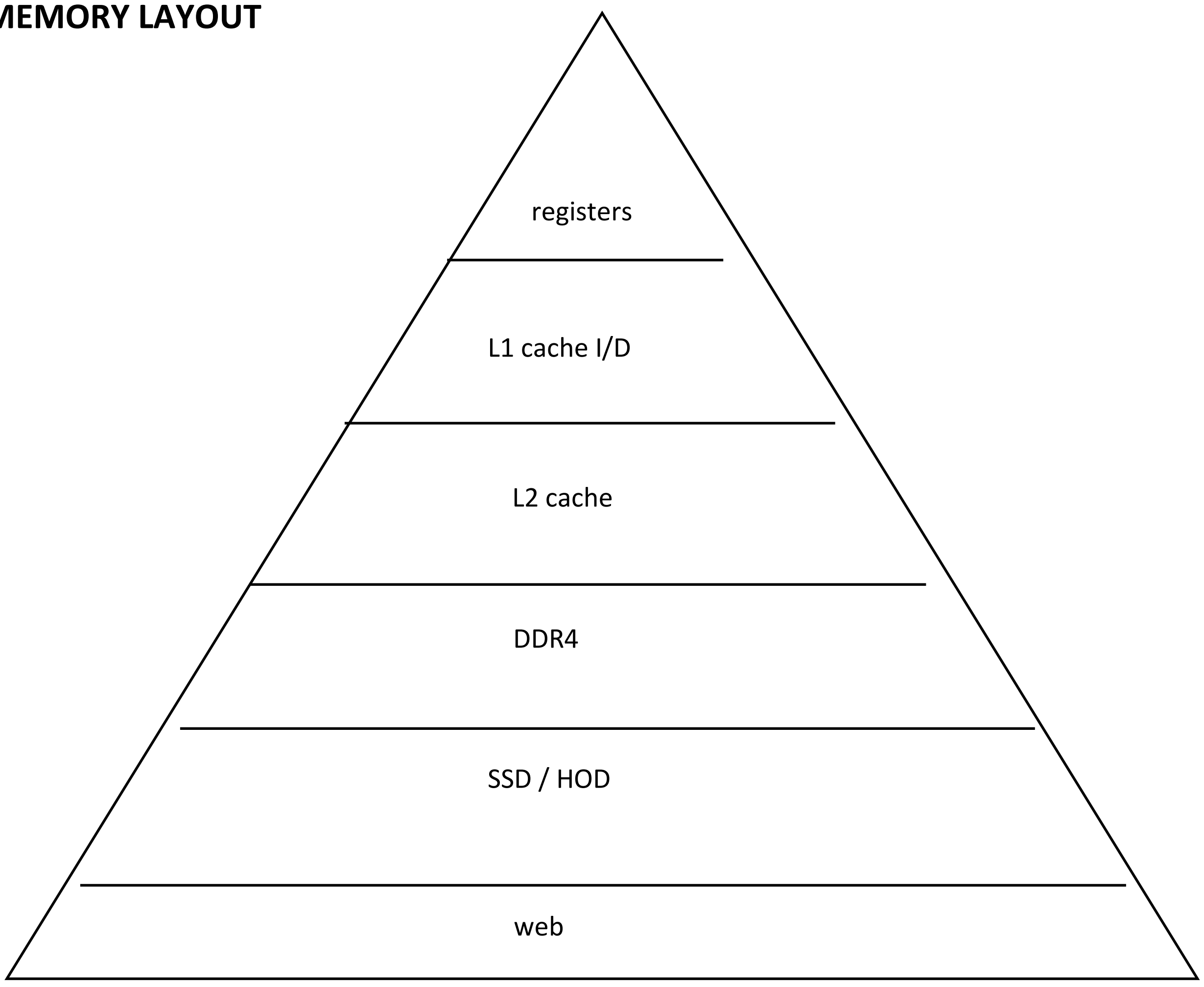
HIT AND MISS LOGIC

- When the processor needs some data, rather than accessing the main memory every time (as it increases the memory access time by a huge amount), it will first search the CACHE.
- If the CACHE contains the data which is being requested, it counts as a HIT.
- Each line in CACHE will have its own tag and the tag from the data(that is being requested), if they match its HIT.
- If the data is not in the CACHE, it’s a MISS.
- The number HITs should always be more than the number of MISS, and that is the goal of a optimizing a CACHE controller.

ARCHITECTURE



MEMORY LAYOUT



WRITE STRATEGIES

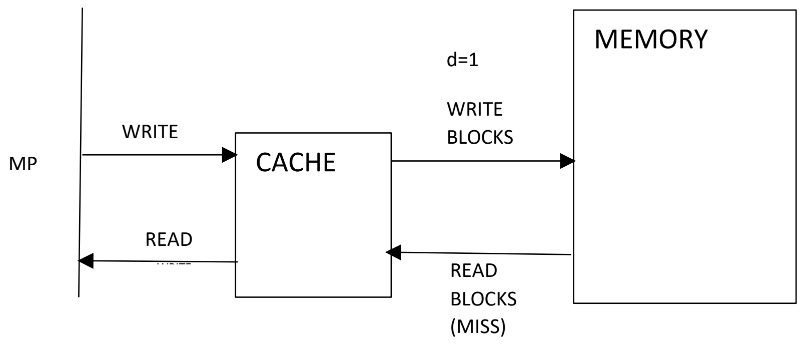
1) WRITEBACK(WB)

write the value to the cache.

int i;
i not in cache :

read i:
miss: find block to replace in line, if d=1 write to memory, read in block with var i.
hit: read i from cache.

Write i:
Write i to cache , mask d=1, update LRU.



2) WRITE THROUGH ALLOCATE(WTA)

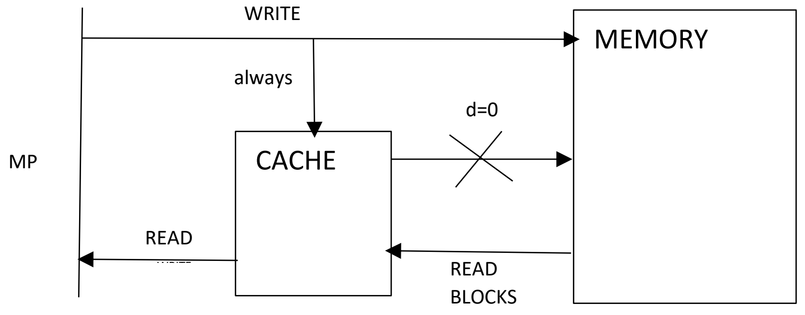
write to cache.
(if not in cache , read from cache, write over it), write to memory.

i not in cache :

read i:
miss: find block to replace in line, if d=1 write to memory, read in block with var i
hit: read i from cache.

```
Write i: write i to memory
        If(I not in cache -miss)
        {
          Find block to replace
          Read in block from memory
```

}
Write i to cache.



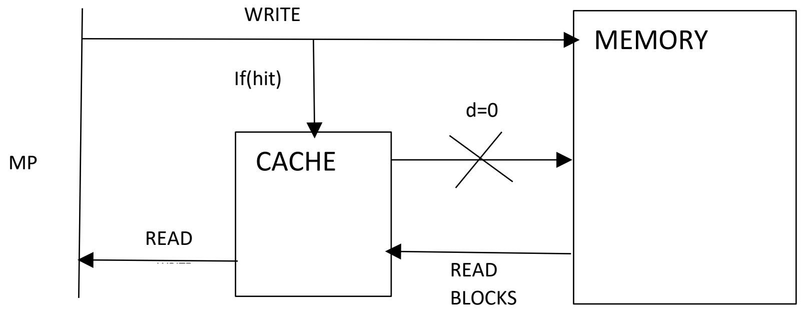
3) WRITE THROUGH NON-ALLOCATE(WTNA)

- write to cache if present, write to memory.

  i not in cache :

  read i:
  miss: find block to replace in line, if d=1 write to memory, read in block with var i
  hit: read i from cache.

  Write: write i to memory, if i in cache (hit), write i to cache.



Write Strategy	Write to Cache	Write to Memory
WB	Always writes to Cache	Never writes to Memory
WTA	Always writes to Cache	Always writes to Memory
WTNA	If hit writes to Cache	Always writes to Memory

ALGORITHM FOR OPERATION OF CACHE

Read Memory:

```
void read_Memory(uint8_t *address, uint8_t byteCount)
{
    int i = 0;
    old_line = -9999;
    int line=0, tag=0;
    rmc++;
    for(i = 0; i<byteCount; i++)
    {
        line = fnLine(address);
        tag = fnTag(address);

        if(line != old_line)
        {
            Read_Block(line, tag);
            old_line = line;
        }
    address++;
    }
}
```

Read Block:

```
void read_Block(uint32_t line, uint32_t tag)
{
    rbc++;
    int hit_LRU, rd_index = 0;
    rd_index = ifCacheHit(line,tag);
    if(rd_index != CACHE_NOT_HIT)
    {
        rbhc++;
    }
    else
    {
        rbmc++;
        rd_index = allBlocksValid(line);
        if(rd_index == ALL_VALID)
        {
            rd_index = oldestLRU(line);
            if(cacheDirty[line][rd_index] == true)
                {
                  rbrdc++;
                  cacheDirty[line][rd_index] = false;
                }
            rbrc++;
        }
        else
            return;


        cacheValid[line][rd_index] = true;
        cacheDirty[line][rd_index] = false;
        cacheTag[line][rd_index] = tag;
        hit_LRU = cacheLRU[line][rd_index];
        update_LRU(line, hit_LRU);
    }
}
```

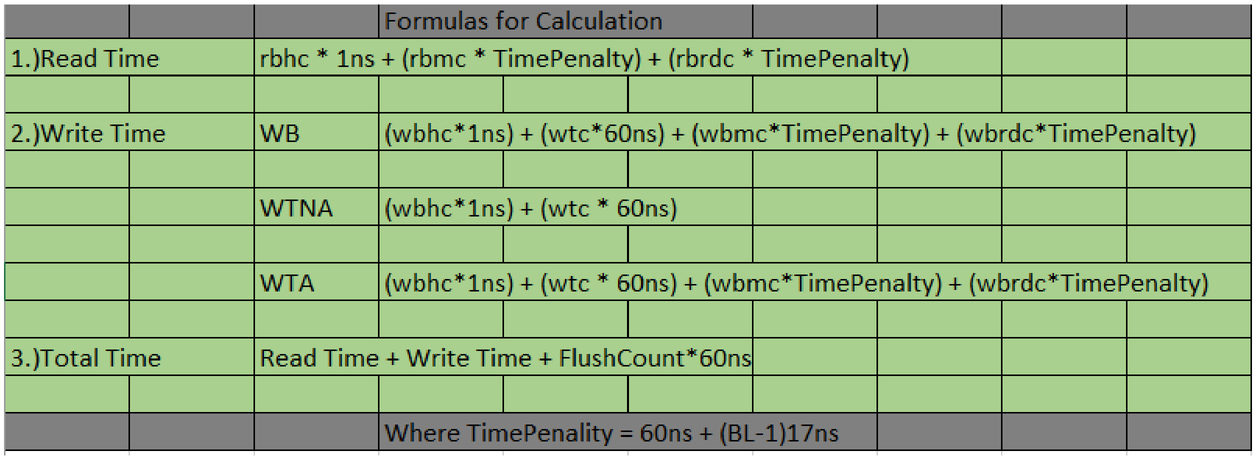
Write Memory:

```
void write_Memory(uint8_t* address, uint8_t byteCount, uint8_t WS)
{
    wmc++;

    int i = 0, line=0, tag=0;
    old_line = -9999;
    for(i = 0; i < byteCount; i++)
    {
         line = fnLine(address);
         tag = fnTag(address);

        if(line != old_line)
        {
            if(( WS == WTA) || (WS == WTNA))
                wtc++;
            write_Block(line, tag, WS);
            old_line = line;
        }
        address++;
    }
}
```

Write Block:

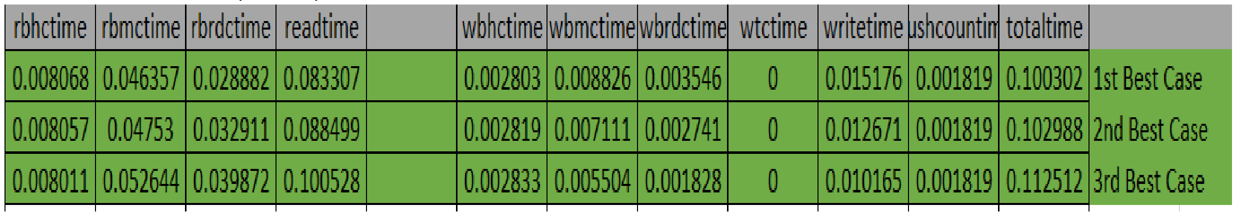
```
void write_Block(uint32_t line, uint32_t tag, uint8_t WS)
{
    int wr_index = 0, hit_LRU;
    wbc++;
    wr_index = ifCacheHit(line,tag);
    if(wr_index != CACHE_NOT_HIT)
    {
        wbhc++;
    }
    else
    {
        wbmc++;
        wr_index = allBlocksValid(line);
        if((wr_index == ALL_VALID ) && !(WS == WTNA))
        {
            wr_index = oldestLRU(line);
            if(cacheDirty[line][wr_index] == true)
                {
                  wbrdc++;
                  cacheDirty[line][wr_index]=false;
                }
            wbrc++;
        }
    }
    if(wr_index!=ALL_VALID)
    {
        hit_LRU = cacheLRU[line][wr_index];
        update_LRU(line, hit_LRU);
        if(WS == WB)
        {
            cacheDirty[line][wr_index] = true;
        }
        cacheValid[line][wr_index] = true;
        cacheTag[line][wr_index] = tag;
    }
}
```

PERFORMACE OUTPUT



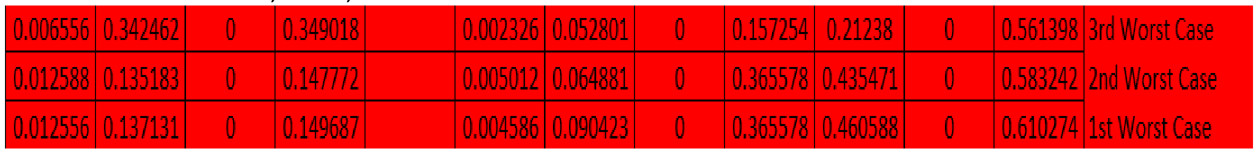
		Formulas for Calculation
1.)Read Time	rbhc * 1ns + (rbmc * TimePenalty) + (rbrdc * TimePenalty)	
2.)Write Time	WB	(wbhc*1ns) + (wtc*60ns) + (wbmc*TimePenalty) + (wbrdc*TimePenalty)
	WTNA	(wbhc*1ns) + (wtc * 60ns)
	WTA	(wbhc*1ns) + (wtc * 60ns) + (wbmc*TimePenalty) + (wbrdc*TimePenalty)
3.)Total Time	Read Time + Write Time + FlushCount*60ns	
		Where TimePenality = 60ns + (BL-1)17ns

1st Best Case: BL = 4, N = 16, WS = 0
2nd Best Case: BL = 4, N = 8, WS = 0
3rd Best Case: BL = 4, N = 4, WS = 0



rbhctime	rbmctime	rbrdctime	readtime		wbhctime	wbmctime	wbrdctime	wtctime	writetime	ushcountin	totaltime	
0.008068	0.046357	0.028882	0.083307		0.002803	0.008826	0.003546	0	0.015176	0.001819	0.100302	1st Best Case
0.008057	0.04753	0.032911	0.088499		0.002819	0.007111	0.002741	0	0.012671	0.001819	0.102988	2nd Best Case
0.008011	0.052644	0.039872	0.100528		0.002833	0.005504	0.001828	0	0.010165	0.001819	0.112512	3rd Best Case

1st Worst Case: BL = 1, N = 2, WS = 2
2nd Worst Case: BL = 1, N = 2, WS = 1
3rd Worst Case: BL = 8, N = 2, WS = 1



0.006556	0.342462	0	0.349018		0.002326	0.052801	0	0.157254	0.21238	0	0.561398	3rd Worst Case
0.012588	0.135183	0	0.147772		0.005012	0.064881	0	0.365578	0.435471	0	0.583242	2nd Worst Case
0.012556	0.137131	0	0.149687		0.004586	0.090423	0	0.365578	0.460588	0	0.610274	1st Worst Case