Assignment 3: Memory Hierarchy

Objective **:** Understanding different types of cache policies

Motivation : Now a day, in a high performance embedded system, cache plays an important role in designing the system. In processor chip increasing the cache size is not always feasible. In embedded processors, for example, cache takes around 30% area of processor, and 40% of power is consumed in cache only. So it is better not to arbitrarily increase the size of the cache, but try to improve performance by using other techniques and policies.

Presently available cache or processor simulators (e.g., simplescalar, Dinero, Cachegrind, Memprof, etc.) do not allow us to simulate all cache r/w fetch policies. This assignment aims at developing a more versatile cache simulator and experimenting with it.

Statement **:**

To simulate different cache policies as mentioned above, you have to design a cache simulator. In this cache simulator you will make modules for cache, bus, memory and processor (request generator). All read/write requests will be queued up in FIFO manner and served by the module at certain rate. As a result, the simulator will report the performance of processor, CPI, for different cache policies, given a cache size. Initially, you will assume that data hazards and control hazards are absent and stalls are only due to memory read/write. Subsequently we will see how this assumption can be relaxed.

Cache policies to be covered:

1. WT, WB,WTWA,WTNA
2. Various block loading policies
3. Various read request policies (concurrent request to cache and memory, Direct Path Processor and Memory)
4. Using prefetch buffer and victim cache
5. Various replacement policies
6. Block size, degree of associativity and cache size would be varied in suitable ranges.

Implementation Hint: C++ and C++ STL library can be a good tool to design this simulator. There are inbuilt data structures like list, queue, vector etc. So they will save lot of extra work. We don't require any thread library to do this work.

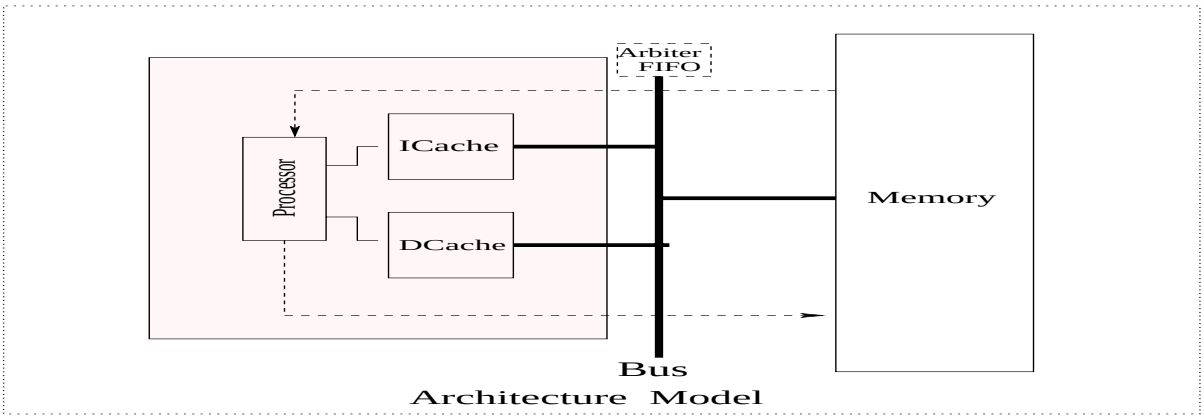
Assumption:

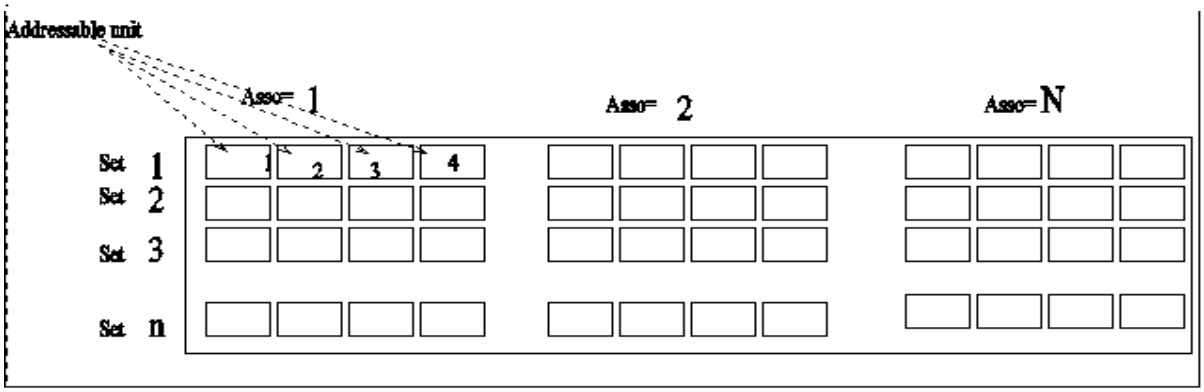
Assume a simple pipeline RISC processor generating instruction read request and data load store request. Processor pipeline will stall for all read request until the getting data/instruction into processor. Processor pipeline will not stall for all write/store request as all store request are queued, but it will effect in bus delay, memory delay and subsequent cache access.

System model is shown in figure

1. n=number of set, N=Associativity, b=block size
2. r= incoming request from processor
3. Cache Size = n x N x b
4. For example, set =8, Asso=2 and b=4 then cachesize=8x2x4=64,
5. MemorySize=M=1024

For any request r where $0<r<1024$, linearly search block (r-r%b) in set r%(Nxb)
For any miss, replace incoming block with a block from corresponding set only using Any replacement policy

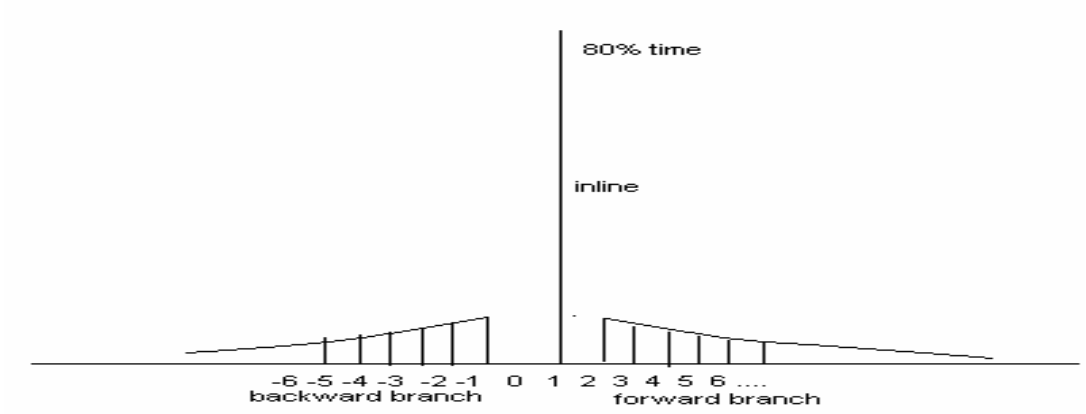




For simplicity clock frequency is same for all, but access time is different for different module in-terms of cycles

Read/write request generator (Processor model): We assume processor will fetch new instruction in each cycle and out of them average 30% are load/store requests. Following algorithm can be used to generate synthetic request in a processor

```
In each cycle
      If( no pending instruction and data read or no-branch) {
       Generate_instruction_read_request();//with distribution given in figure
       Generate_a_data_read_write_request();// with prob=0.3;
}
```



```
int Generate_instruction_read_request(){
                        if( (rand()%100) < 80 ) //.8 prob
                                new_request = old_request+1; //Inline
                        else //branch is  .2 prob
                                {
                                 r=0;
                                while(r!=0){
                                      x=rand()%M;//M is Max distance of branch
                                      if( rand()%100 < (( 1- exp(x-M))*100 )
                                           if (rand()%2 ==0) r=x;
                                            else r=-x;
                                     }
                                r = r+ old_request;
                                }
  }
int Generate_a_data_read_write_request(){
                    if((rand()%100) < 30) // 0.3 prob it generate load
                           {
                           if( (rand()%100) < 70 ) //.7 prob
                                  new_request = old_request+1; //next data
                           else //access to other location is  .3 prob
                                  {
                                   r=0;
                                  while(r!=0){
                                        x=rand()%M;
                                        if( rand()%100 < (( 1- exp(x-M))*100 )
                                             if (rand()%2 ==0) r=x; else r=-x;
                                       }
                                  r = r+ old_request;
                                  }
                           if((rand()%100) < p*10 ) // p prob WRITE
                                  ST=1; //Request is ST
                            else  LD=1;
                           }

                    else return -1; //No LOAD or STORE
}
```

Memory model:
I-cache/ D-cache access time = 1 cycle
Main Memory: Access to 1st word 10 cycles and subsequent word is 3cycle,
Keep all these latency variable (or in config file, so that can be changed later).

Optionally you can model in DRAM memory, Let DRAM size is NxN then it contain a buffer of size N,
If access word/block is in buffer then it will take less time, then if it is not in buffer.

Deadline : 1 April 2006.

Following are intermediate milestones
Assignment up on 3rd March
Design of the simulator : 10th March
Implementation: 24th March
Simulation result for each cache policy (given above) : 1st April