TEST OF HIGH PERFORMANCE COMPUTER ARCHITECTURE 31-01-2011 **MATR.NO.**____________________

REVISION 14-11-2025 **SURNAME**______________________

NAME______________________

1) (26/40) Let us consider a Directory-Based Invalidation Coherence Protocol for a 64-core system. In order to implement the sharer list (S, list of cores sharing a block of data) in the HOME core, a per-block bit-vector is maintained. The list S tracks all cores that share the associated data block. The k-th bit of the bit vector indicates whether or not the corresponding k-th core has a copy of the block (bit==1 → has the block, bit==0→ the core does not have a copy the block). Assume that the block-size is 32 bytes.

Question 1a) Calculate the cost of the "bit-vector" solution as a fraction of the bits that are needed to store the data. Perform the calculation for the cases of a 64-core machine and a 4-core machine.

Question 1b) Assume that the S-list is replaced by an optimized solution ("Single-Sharer") that uses, instead of the S-list, a single 6-bit identifier (SID) which indicates which a core (the last one) which has a copy of the block. When there is a Bus-Read Transaction involving a certain block, the copy of that block in the core indicated by SID has to be invalidated (the HOME core initiates an invalidation request, then the current sharer core sends an acknowledge message to the HOME core; the HOME core then replaces the SID content with the core ID of the new sharer core and sends an acknowledge message to the new sharer core). Compare the efficiency of the "Bit-Vector" solution with the "Single-sharer" solution by filling up the first two empty columns in the following tables with the number of invalidation requests that are generated by the coherence protocol at each step. Assume that the data block B is not initially present in any cache.

SEQUENCE 1	Bit-Vector -- # of invalidations	Single-Sharer -- # of invalidations	Single-Sharer+Global-Bit -- # of invalidations
STEP1: Core #0 reads B	0	0	
STEP2: Core #1 reads B			
STEP3: Core #0 reads B			

SEQUENCE 2	Bit-Vector -- # of invalidations	Single-Sharer -- # of invalidations	Single-Sharer+Global-Bit -- # of invalidations
STEP1: Core #0 reads B	0	0	
STEP2: Core #1 reads B			
STEP3: Core #2 reads B			

Question 1c) Assume that the previous solution ("Single-Sharer") is improved by adding a global bit G besides the S-ID ("Single-Sharer + Global-Bit"). This global bit is set to 1 to indicate that more than one core has a copy of the block. When G==1, the HOME core does not track any more the specific sharer and it assumes that all cores could potentially share the block. Fill-up the third empty column in the previous table.

Question 1d) Describe with a state diagram the complete functioning of the three coherence protocol variants of the cases 1a, 1b, 1c above, assuming three possible status conditions for the block: U=Uncached (or invalid), EM=Exclusive-or-Modified (or owned, which means not shared and possibly modified), S=Shared (Shared, which is also clean, i.e., coherent with the memory).
The possible transactions regard a Read-Block, Invalidation and Flush (or Write-back) which implies to evict the block from the cache. Use this notation: L=Sharer-List, Cx=Previous Core doing an operation on a block, Cy=Current Core doing an operation on a block.

2) (13/40) Write a simple program in the CUDA programming model in order to add two vectors A and B and obtain a third vector C. Each vector has a size of 100'000 floating point elements.

TEST OF HIGH PERFORMANCE COMPUTER ARCHITECTURE 31-01-2011

REVISION 14-11-2025

SOLUTION TRACE

1a)
For the 4-core system: 4bit / 32 Byte = 4 / (32*8) = 1/64 = 0.015625 (about 1.6%)
For the 64-core system: 64bit / 32 Byte = 64 / (32*8) = 1/4 = 0.25 (25%)

1b) and 1c)

SEQUENCE 1	Bit-Vector -- # of invalidations	Single-Sharer -- # of invalidations	Single-Sharer+Global-Bit -- # of invalidations
STEP1: Core #0 reads B	0	0	0
STEP2: Core #1 reads B	0	1	0
STEP3: Core #0 reads B	0	1	63

- For the Bit-Vector solution: no invalidation is sent.
- For the Single-Sharer solution: 1 invalidation goes to Core-0 when Core-1 reads B; 1 invalidation goes to Core-1 when Core-0 reads B.
- For the Single-Sharer+Global-Bit: there are 63 invalidations by avoiding that the HOME Core invalidates also Core-0, the one which sent a message for invalidation to the HOME core.

SEQUENCE 2	Bit-Vector -- # of invalidations	Single-Sharer -- # of invalidations	Single-Sharer+Global-Bit -- # of invalidations
STEP1: Core #0 reads B	0	0	0
STEP2: Core #1 reads B	0	1	0
STEP3: Core #2 reads B	0	1	63

- For the Bit-Vector solution: no invalidation is sent.
- For the Single-Sharer solution: 1 invalidation goes to Core-0 when Core-1 reads B; 1 invalidation goes to Core-1 when Core-2 reads B.
- For the Single-Sharer+Global-Bit: there are 63 invalidations by avoiding that the HOME Core invalidates also Core-2, the one which sent a message for invalidation to the HOME core.

TEST OF HIGH PERFORMANCE COMPUTER ARCHITECTURE 31-01-2011

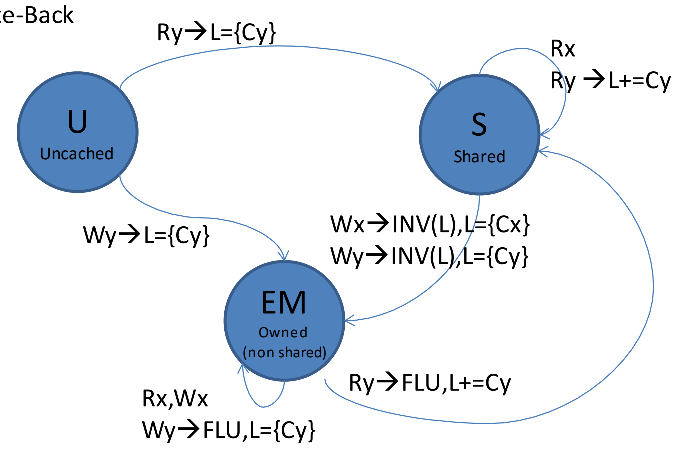
REVISION 14-11-2025

SOLUTION TRACE

1d) The state diagram should represent the state of the block (not the location Home/Local/Remote).
The protocol is Write-Invalidate.

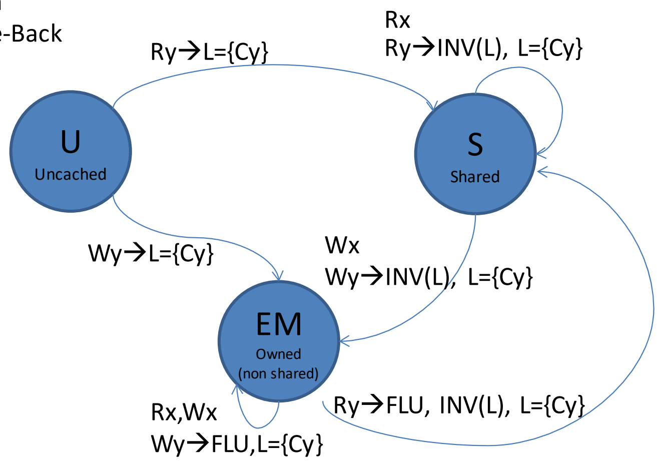
Bit-Vector

L=Sharer list
Rx=Read by Cx
Wx=Write by Cx
INV=Invalidation
FLU=Flush/Write-Back
X=Prev. Core
Y=Current Core



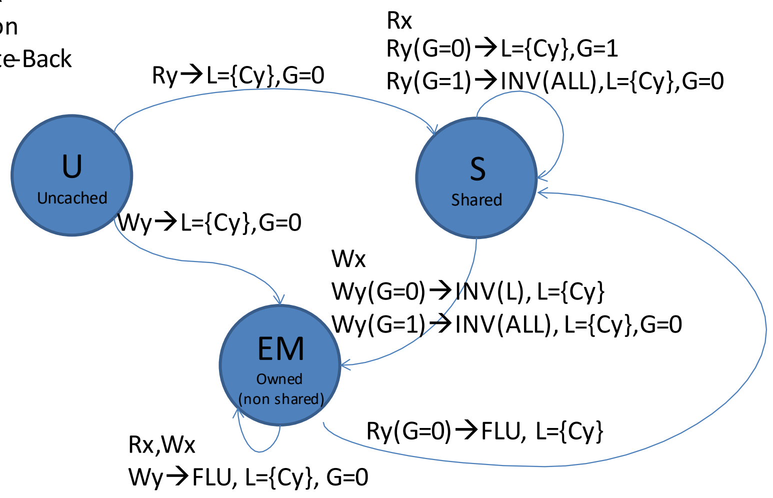
Single-Sharer

S=Sharer list
Rx=Read by Cx
Wx=Write by Cx
INV=Invalidation
FLU=Flush/Write-Back
X=Prev. Core
Y=Current Core



Single-Sharer+Global-bit

S=Sharer list
Rx=Read by Cx
Wx=Write by Cx
INV=Invalidation
FLU=Flush/Write-Back
X=Prev. Core
Y=Current Core



TEST OF HIGH PERFORMANCE COMPUTER ARCHITECTURE 31-01-2011

REVISION 14-11-2025

SOLUTION TRACE

2) A possible solution is:

```
#define  N 100000

__global__  void vecAdd(float* A, float* B, float* C)
{
   int i = threadIdx.x + blockIdx.x * blockDim.x;
   if (i < N) C[i] = A[i] + B[i];
}

int  main()
{
    float A[N], B[N], C[N];
    float *devPtrA, *devPtrB, *devPtrC;

    int memsize= N * sizeof(float);

    cudaMalloc((void**)&devPtrA, memsize);
    cudaMalloc((void**)&devPtrB, memsize);
    cudaMalloc((void**)&devPtrC, memsize);
    cudaMemcpy(devPtrA, A, memsize, cudaMemcpyHostToDevice);
    cudaMemcpy(devPtrB, B, memsize, cudaMemcpyHostToDevice);

    vecAdd<<<(N+127)/128, 128>>>(devPtrA,  devPtrB, devPtrC);

    cudaMemcpy(C, devPtrC, memsize, cudaMemcpyDeviceToHost);

    cudaFree(devPtrA);
    cudaFree(devPtrB);
    cudaFree(devPtrC);

   // USO DEL VETTORE C

}
```