HIGH PERFORMANCE COMPUTER ARCHITECTURE final exam 18-12-2019 **MATR.NO.**____________________

SURNAME____________________

FIRST NAME____________________

1) (POINTS 25/40) Consider a four-processor bus-based multiprocessor using the MSI protocol. Each processor executes a TAS instruction to lock and gain access to an empty critical section. The initial condition is such that processor 1 has the lock and processor 2, 3, and 4 are spinning on their caches waiting for the lock to be released. Every processor gets the lock once and exits the program. These are the implementations of the lock and unlock:

```
Lock:   lw R1, mylock      # R1 = &mylock
        bne R1, R0, Lock   # if (R1 != 0) jump to Lock
        TAS R1, mylock     # atomically_do {R1 = &mylock; mylock = 1;}
        bne R1, R0, Lock   # if (R1 != 0) jump to Lock
        ret

Unlock: sw 0, mylock       # write 0 into &mylock
        ret
```

Note1: the semantic of the TAS (Test And Set) instruction is the following: atomically reads the specified memory location (mylock) and writes a one into that memory location (mylock). Note2: this implementation of the Lock tries to minimize the probability to have the bus locked by the TAS (this implementation is also known as Test-and-Test-and-Set). Note3: the lock is closed when mylock==1 and it is open when mylock==0.

By using the following tables, show the operations and bus transactions (or comments): A) in the best case (least number of transactions) and B) in the worst case (highest number of transactions)

A) Best case:

Bus Trans. Number	Processor Operation	P1	P2	P3	P4	Bus Transactions/Comments
---	(Init.state)	S	S	S	S	Initially, P1 holds the lock
1	sw1	M	I	I	I	**BusUpgr** – P1 releases the lock

B) Worst case:

Bus Trans. Number	Processor Operation	P1	P2	P3	P4	Bus Transactions/Comments
---	(Init.state)	S	S	S	S	Initially, P1 holds the lock
1	sw1	M	I	I	I	**BusUpgr** – P1 releases the lock

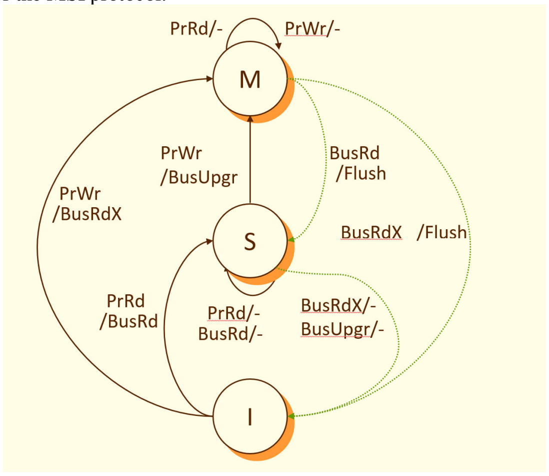
2) (POINTS 15/40) Write a CILK function that reads a color array (int color[1024]) and writes an array “int histogram[256]” that contains the frequency of each of 256 possible colors (the 256 values are the value that each element of color[] can assume). A serial or serialized version **has to be avoided**. The program should be written in a way that it exploits Thread Level Parallelism as offered by CILK. Reference scalar version:

```
typedef unsigned int uint;
typedef unsigned char uchar;
#define HISTOGRAM_BIN_COUNT 256
uchar Color[1024];
uint Histogram[HISTOGRAM_BIN_COUNT];
void histo_scalar(uint *histogram, uchar *color, uint size) {
    for(uint i=0; i<size; i++ ) histogram[ color[i] ]  += 1;
}
```

Hints: Use cilk_spawn, cill_sync, cilk_for as appropriate, try to perform operations in a hierarchical way. For atomic accesses you can use: pthread_mutex_t lock [256] and then pthread_mutex_lock(&lock[…]) and pthread_mutex_unlock(&lock[…]) as appropriate.

HIGH PERFORMANCE COMPUTER ARCHITECTURE final exam 18-12-2019
(solution trace)

1) Remembering the state diagram for the MSI protocol:



1A) The best case happens if the interleaving of the operations is such that each processor attempts and get access to the critical section one after the other.

Bus Trans. Number	Processor Operation	P1	P2	P3	P4	Bus Transactions/Comments
---	(Init.state)	S	S	S	S	Initially, P1 holds the lock
1	sw1	M	I	I	I	**BusUpgr** – P1 releases the lock
2	lw2	S	S	I	I	**BusRd/Flush** –P2 reads the lock
3	TAS2	I	M	I	I	**BusUpgr** – P2 tries to lock and succeeds
--	sw2	I	M	I	I	P2 releases the lock
4	lw3	I	S	S	I	**BusRd/Flush** –P3 reads the lock
5	TAS3	I	I	M	I	**BusUpgr** – P3 tries to lock and succeeds
--	sw3	I	I	M	I	P3 releases the lock
6	lw4	I	I	S	S	**BusRd/Flush** –P4 reads the lock
7	TAS4	I	I	I	M	**BusUpgr** – P4 tries to lock and succeeds
--	sw4	I	I	I	M	P4 releases the lock

1B) The worst case happens if the interleaving of the operations is such that each processor attempts simultaneously the “lw” to read the status of mylock and then simultaneously try to get the access through the TAS instruction.

Bus Trans. Number	Processor Operation	P1	P2	P3	P4	Bus Transactions/Comments
---	(Init.state)	S	S	S	S	Initially, P1 holds the lock
1	sw1	M	I	I	I	**BusUpgr** -- P1 releases the lock
2	lw2	S	S	I	I	**BusRd/Flush** – P2 reads the lock
3	lw3	S	S	S	I	**BusRd** – P3 reads the lock
4	lw4	S	S	S	S	**BusRd** – P4 reads the lock
5	TAS2	I	M	I	I	**BusUpgr** – P2 gets the lock
6	TAS3	I	I	M	I	**BusRdX/Flush** no lock
7	TAS4	I	I	I	M	**BusRdX/Flush** no lock
8	st2	I	M	I	I	**BusRdX/Flush** -- P2 releases the lock
9	lw3	I	S	S	I	**BusRd/Flush** – P3 reads the lock
10	lw4	I	S	S	S	**BusRd** – P4 reads the lock
11	TAS3	I	I	M	I	**BusUpgr** – P3 gets the lock
12	TAS4	I	I	I	M	**BusRdX/Flush** no lock
13	sw3	I	I	M	I	**BusRdX/Flush** -- P3 releases the lock
14	lw4	I	I	S	S	**BusRd/Flush** – P4 reads the lock
15	TAS4	I	I	I	M	**BusUpgr** – P4 gets the lock
---	sw4	I	I	I	M	P4 releases the lock

* Depending on the implementation a BusRdX could be directly associated to the (atomic) TAS instruction.

2) This is the CILK code for a possible implementation of the requested kernel:

```
typedef unsigned int uint;
typedef unsigned char uchar;
#define HISTOGRAM_BIN_COUNT 256
uchar Color[1024];
uint Histogram[HISTOGRAM_BIN_COUNT];

#define Cilk_lockvar pthread_mutex_t
#define Cilk_lock pthread_mutex_lock
#define Cilk_unlock pthread_mutex_unlock
Cilk_lockvar lock[HISTOGRAM_BIN_COUNT];
```

```
void histo_cilk2(uint *histogram, uchar *color, uint size)
{
    if (size == 1) {
        Cilk_lock(&lock[*color]);
        histogram[*color]++;
        Cilk_unlock(&lock[*color]);
    }
    else {
        cilk_spawn histo_cilk2(histogram, color, size/2);
        cilk_spawn histo_cilk2(histogram, color + size/2, size - size/2);
        cilk_sync;
    }
}
```