HIGH PERFORMANCE COMPUTER ARCHITECTURE final exam 23-12-2020 **MATR.NO.**_____

SURNAME_____

FIRST NAME_____

1) (POINTS 25/40) 1) Let's consider a shared-bus shared-memory coherence protocol with the following characteristics:
- Processor Operations: Read (**PrRd**) and Write (**PrWr**).
- Bus Transactions: Read a data-block from Memory (**BusRd**), Propagate a PrWr on the bus (**BusUpd** – a single data-word goes to other caches and to memory), Write back a block onto the bus (**Flush** – the data block goes to memory).
- There are 4 states: 1) the copy is not valid (**I** – Invalid); 2) the copy is in one cache only and we assume that it has been modified even if it has been just loaded (**D** – Dirty); 3) the copy has been just loaded in the current cache and other copies exist in other caches (**S0** – shared and no other processor has updated it after loading the copy); 4) the copy is shared but some other processor had issued a write into that block (**S1** – shared and other processors issued exactly one update to that block).
- The bus has a shared line, which is activated by remote snoopers in response to a bus transaction to indicate whether (S) or not (S') other copies of the addressed block exist in other caches.

Assume that this protocol is used in a four-processor system, where 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. Also assume that NO write is propagated on the bus if the TAS finds a closed lock (i.e., if the TAS fails).

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

Case A:

Bus Trans. Number	Processor Operation	P1	P2	P3	P4	Bus Transactions/Comments
---	(Init.state)	S0	S0	S0	S0	Initially, P1 holds the lock
1	sw1	S0	S1	S1	S1	**BusUpd** – P1 releases the lock

Case B:

Bus Trans. Number	Processor Operation	P1	P2	P3	P4	Bus Transactions/Comments
---	(Init.state)	S0	S0	S0	S0	Initially, P1 holds the lock
1	sw1	S0	S1	S1	S1	**BusUpd** – P1 releases the lock

2) (POINTS 15/40) Write an MPI 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 parallelism as offered by MPI. Reference scalar version:

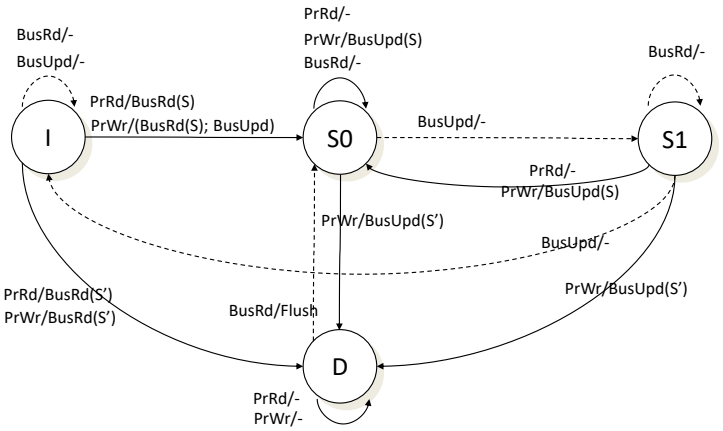
```
unsigned char Color[1024]; int Histogram[256];
void histo_scalar(int *histogram, unsigned char *color, int size, int wid, int nw) {
    for(int i=0; i<size; i++ ) histogram[ color[i] ]  += 1;
}
```

Hints: use send and receive to distribute the work among the available workers/nodes; use MPI primitives: **`MPI_Send, MPI_Recv, MPI_Iprobe`**.

HIGH PERFORMANCE COMPUTER ARCHITECTURE final exam 23-12-2020

(solution trace) REV.231123

1) The state diagram for this protocol:



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)	S0	S0	S0	S0	Initially, P1 holds the lock
1	sw1	S0	S1	S1	S1	**BusUpd1** – P1 releases the lock
---	lw2	S0	S0	S1	S1	P2 reads the lock (and it finds it open, i.e. ==0)
2	TAS2	S1	S0	I	I	**BusUpd2** – P2 tries to lock and succeeds
3	sw2	I	D	I	I	**BusUpd2** – P2 releases the lock
4	lw3	I	S0	S0	I	**BusRd3+Flush2** –P3 reads the lock (and it finds it open, i.e. ==0)
5	TAS3	I	S1	S0	I	**BusUpd3** – P3 tries to lock and succeeds
6	sw3	I	I	D	I	**BusUpd3** – P3 releases the lock
7	lw4	I	I	S0	S0	**BusRd4+Flush3** –P4 reads the lock (and it finds it open, i.e. ==0)
8	TAS4	I	I	S1	S0	**BusUpd4** – P4 tries to lock and succeeds
9	sw4	I	I	I	D	**BusUpd4** – P4 releases the lock

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. In this cache each processor has cached the lock and if the TAS fails no bus transaction is issued

Bus Trans. Number	Processor Operation	P1	P2	P3	P4	Bus Transactions/Comments
---	(Init.state)	S0	S0	S0	S0	Initially, P1 holds the lock
1	sw1	S0	S1	S1	S1	**BusUpd1** -- P1 releases the lock
---	lw2	S0	S0	S1	S1	P2 reads the lock → reads 0 → the first bne doesn't branch
---	lw3	S0	S0	S0	S1	P3 reads the lock → reads 0 → the first bne doesn't branch
---	lw4	S0	S0	S0	S0	P4 reads the lock → reads 0 → the first bne doesn't branch
2	TAS2	S1	S0	S1	S1	**BusUpd2** – P2 gets the lock and updates the others
3	TAS3	I	S1	S0	I	TAS fails: no lock → **BusUpd3**
4,5	TAS4	I	I	S1	S0	**BusRd4** - TAS fails: no lock → **BusUpd4**
6,7	sw2	I	S0	I	S1	**BusRd2+BusUpd2** -- P2 releases the lock
9	lw3	I	S0	S0	S1	**BusRd3** - P3 reads the lock
---	lw4	I	S0	S0	S0	P4 reads the lock
10	TAS3	I	S1	S0	S1	**BusUpd3** – P3 gets the lock and updates the others
---	TAS4	I	I	S1	S0	TAS fails: no lock
11	sw3	I	I	S0	S1	**BusUpd3** -- P3 releases the lock
---	lw4	I	I	S0	S0	P4 reads the lock
12	TAS4	I	I	S1	S0	**BusUpd4** – P4 gets the lock and updates the others
13	sw4	I	I	I	D	**BusUpd4** – P4 releases the lock

2) This is the MPI code for a possible implementation of the requested function:

```
#define MASTER 0
#define TAG_GENERAL 1
void histo_mpi(int *histogram, char *color, int size, int wid, int nw) {
    int i, histogram_private[256];
    for(i=0; i<256; i++) histogram_private[i] = 0;
    MPI_Status Stat;
    int ssize = (size - 1) /nw + 1;
    char *slice = malloc(ssize * sizeof(char));

    if (wid == MASTER) { // Distribute the work
        for (int i= 0; i<ssize; ++i) slice[i] = color[i]; // Assign 1st slice to master
        for(int w=1; w<nw; ++w)  // Send the other slices to the slaves
            MPI_Send(color + w*ssize, ssize, MPI_CHAR, w, TAG_GENERAL, MPI_COMM_WORLD);
    } else { // slave
        int dataWaitingFlag; // Wait until a message is there to be received
        do MPI_Iprobe(MASTER, TAG_GENERAL, MPI_COMM_WORLD,
                      &dataWaitingFlag, MPI_STATUS_IGNORE);
        while (!dataWaitingFlag);
        MPI_Recv(slice, ssize, MPI_CHAR, MASTER, TAG_GENERAL, MPI_COMM_WORLD, &Stat);
    }

    // Processing data
    for (int i= 0; i<ssize; ++i) histogram_private[slice[i]]++; free(slice);

    if (wid == MASTER) { // Process the partial results
        int w, done = 0;

        // Accumulate the result
        for (int i= 0; i<256; ++i) histogram[i] += histogram_private[i];

        do {// Get partial histograms from slaves
            for (w=1; w<nw; ++w) { // round robin check
                int dataWaitingFlag;
                MPI_Iprobe(w, TAG_GENERAL, MPI_COMM_WORLD,
                           &dataWaitingFlag, MPI_STATUS_IGNORE);
                if (dataWaitingFlag) { // Get the message
                    MPI_Recv(histogram_private, 256, MPI_INT, w,
                             TAG_GENERAL, MPI_COMM_WORLD, &Stat);
                    ++done;

                    // Accumulate the result
                    for (int i= 0; i<256; ++i) histogram[i] += histogram_private[i];
                }
            }
        } while (done < nw - 1);
    } else // slave: send back the partial result
        MPI_Send(histogram_private, 256, MPI_INT, MASTER, TAG_GENERAL, MPI_COMM_WORLD);
}
```

Alternatively, by using Scatter/Reduce:

```
void histo_mpi(int *histogram, char *color, int size, int wid, int nw) {
     int rc, i, histogram_private[HISTOGRAM_BIN_COUNT];
    for(i=0; i<HISTOGRAM_BIN_COUNT; i++) histogram_private[i] = 0;
    int ssize = (size - 1) /nw + 1;
    char *slice = malloc(ssize * sizeof(char));

MPI_Scatter(color, size, MPI_CHAR, slice, ssize, MPI_CHAR, 0, MPI_COMM_WORLD);
// Processing data
for (int i= 0; i<ssize; ++i) histogram_private[slice[i]]++; free(slice);
for(int i=0; i<HISTOGRAM_BIN_COUNT; i++)
   MPI_Reduce(&histogram_private[i],
              &histogram[i], 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
}
```