CSE 661

Parallel and Vector Architectures

Midterm Exam – Fall 2007

Tuesday, April 4, 2007

7:00 – 9:30 pm

Computer Engineering Department

College of Computer Sciences & Engineering

King Fahd University of Petroleum & Minerals

Student Name: SOLUTION

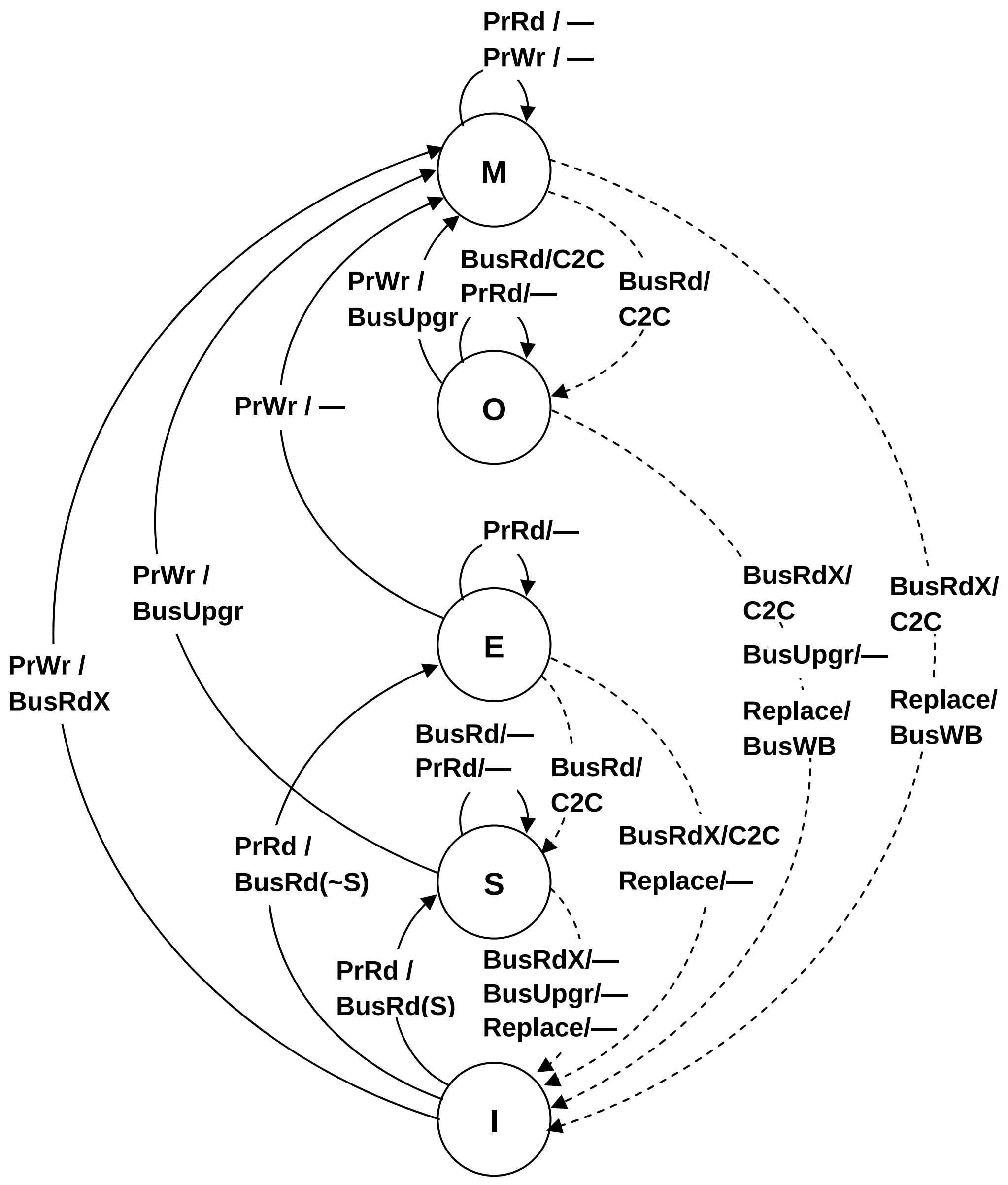
Student ID: ____________________

Q1	/ 20	Q2	/ 35
Q3	/ 25	Q4	/ 25
Total	/ 100		

Q1. (20 pts) In the Illinois MESI cache coherence protocol, when a block is modified in one cache and there is a Bus Read request from a second cache on a read miss, the cache block is flushed on the bus by the first cache. The data is picked up by the requesting cache and is written as well to memory. The cache block is now shared in both caches and memory is up-to-date.

The problem here is that main memory is much slower than the requesting cache. Cache-to-cache transfer is very desirable, but we should avoid updating memory. Therefore, we need to modify the MESI protocol to become MOESI, where a fifth state (O = Owned) is introduced. The Owned state is a shared modified state indicating that memory is NOT up-to-date, but will be updated on a writeback transaction when the block in the Owned state is replaced inside the cache.

Draw the MOESI state diagram, showing all transitions on processor read (PrRd), processor write (PrWr), on a block replacement (Replace), as well as observing a bus read (BusRd), bus read exclusive (BusRdX), and bus upgrade (BusUpgr). Cache-to-Cache transfer (C2C) should be supported only, but Not cache flushing. Bus writeback (BusWB) should happen only when a modified or owned block is replaced.



Q2. (35 pts) Consider the execution of a parallel loop on a bus-based multiprocessor with 2 processors. Processor P0 executes the even iterations and Processor P1 executes the odd one. For each iteration, each processor reads elements **A[i]** and **B[i]**, does the addition, and then writes element **A[i]**, generating two memory reads and one memory write.

```
Original Loop:
for (i=0; i<N; i++) A[i] = A[i] + B[i];

Processor P0 executes even iterations:
for (i=0; i<N; i=i+2) A[i] = A[i] + B[i];

Processor P1 executes odd iterations:
for (i=1; i<N; i=i+2) A[i] = A[i] + B[i];
```

Each processor has a private data cache with **32-byte blocks**. Each cache block can fit **8 array elements**, where each element is 4 bytes. Assume a cold start. The caches blocks are initially invalid forcing initial cache misses to read the blocks from memory.

a) (10 pts) Assuming that the caches implement the **Illinois MESI coherence protocol**, consider the **worst-case** scenario that results in the **worst number of bus transactions**. Complete the table (next page) showing the execution of 5 iterations in P0 and in P1. Compute the bus transactions issued by P0 and P1 for all **N** iterations, where **N** is even.

b) (10 pts) Repeat part **(a)**, assuming that the caches now implement the **Dragon update coherence protocol**. Complete the table showing the execution of 5 iterations in P0 and in P1. Compute the number of bus transactions issued by P0 and P1 for all **N** iterations.

c) (15 pts) Modify the code executed by P0 and P1 to minimize the cache misses and bus transactions. Complete the table showing the execution of 2 iterations only in P0 and in P1 for the MESI and Dragon coherence protocols. Re-compute the total bus transactions issued by P0 and P1 for all **N** iterations for the MESI and Dragon protocols.

Answer the three parts of this question on the next three pages.

Part (a) Solution: MESI Cache Coherence Protocol

Read/Write Operation	Bus Transaction	P0 Cache State	P1 Cache State
Initial Cold Start		I	I
P0 reads A[0]	Bus Read	E	I
P1 reads A[1]	Bus Read	S	S
P0 reads B[0]	Bus Read	E	I
P1 reads B[1]	Bus Read	S	S
P0 writes A[0]	Bus Upgrade	M	I
P1 writes A[1]	Bus Read X, Flush	I	M
P0 reads A[2]	Bus Read, Flush	S	S
P1 reads A[3]		S	S
P0 reads B[2]		S	S
P1 reads B[3]		S	S
P0 writes A[2]	Bus Upgrade	M	I
P1 writes A[3]	Bus Read X, Flush	I	M
P0 reads A[4]	Bus Read, Flush	S	S
P1 reads A[5]		S	S
P0 reads B[4]		S	S
P1 reads B[5]		S	S
P0 writes A[4]	Bus Upgrade	M	I
P1 writes A[5]	Bus Read X, Flush	I	M
P0 reads A[6]	Bus Read, Flush	S	S
P1 reads A[7]		S	S
P0 reads B[6]		S	S
P1 reads B[7]		S	S
P0 writes A[6]	Bus Upgrade	M	I
P1 writes A[7]	Bus Read X, Flush	I	M
P0 reads A[8]	Bus Read (new block)	E	I
P1 reads A[9]	Bus Read (new block)	S	S
P0 reads B[8]	Bus Read (new block)	E	I
P1 reads B[9]	Bus Read (new block)	S	S
P0 writes A[8]	Bus Upgrade	M	I
P1 writes A[9]	Bus Read X, Flush	I	M

Total number of bus transactions (all iterations) =

15 bus transactions (to process 8 elements) × N/8 = 15 N / 8

Part (b) Solution: Dragon Cache Coherence Protocol

Read/Write Operation	**Bus Transaction**	**P0 Cache State**	**P1 Cache State**
Initial Cold Start		X	X
P0 reads A[0]	Bus Read	E	X
P1 reads A[1]	Bus Read	Sc	Sc
P0 reads B[0]	Bus Read	E	X
P1 reads B[1]	Bus Read	Sc	Sc
P0 writes A[0]	Bus Update	Sm	Sc
P1 writes A[1]	Bus Update	Sc	Sm
P0 reads A[2]		Sc	Sm
P1 reads A[3]		Sc	Sm
P0 reads B[2]		Sc	Sc
P1 reads B[3]		Sc	Sc
P0 writes A[2]	Bus Update	Sm	Sc
P1 writes A[3]	Bus Update	Sc	Sm
P0 reads A[4]		Sc	Sm
P1 reads A[5]		Sc	Sm
P0 reads B[4]		Sc	Sc
P1 reads B[5]		Sc	Sc
P0 writes A[4]	Bus Update	Sm	Sc
P1 writes A[5]	Bus Update	Sc	Sm
P0 reads A[6]		Sc	Sm
P1 reads A[7]		Sc	Sm
P0 reads B[6]		Sc	Sc
P1 reads B[7]		Sc	Sc
P0 writes A[6]	Bus Update	Sm	Sc
P1 writes A[7]	Bus Update	Sc	Sm
P0 reads A[8]	Bus Read (new block)	E	X
P1 reads A[9]	Bus Read (new block)	Sc	Sc
P0 reads B[8]	Bus Read (new block)	E	X
P1 reads B[9]	Bus Read (new block)	Sc	Sc
P0 writes A[8]	Bus Update	Sm	Sc
P1 writes A[9]	Bus Update	Sc	Sm

Total number of bus transactions (all iterations) =

12 bus transactions (to process 8 elements) × N/8 = 12 N / 8

Part (c) Solution:

Modified Code for P0 and P1

```
Processor P0 executes first N/2 iterations:
for (i=0; i<N/2; i++) A[i] = A[i] + B[i];
```

```
Processor P1 executes last N/2 iterations:
for (i=N/2; i<N; i++) A[i] = A[i] + B[i];
```

Executing 2 iterations in P0 and in P1 using the MESI and Dragon Coherence Protocols

	MESI Coherence Protocol			**Dragon Coherence Protocol**		
Read/Write Operation	**Bus Transaction**	**P0 Cache State**	**P1 Cache State**	**Bus Transaction**	**P0 Cache State**	**P1 Cache State**
Cold Start		I	I		X	X
P0 reads A[0]	Bus Read	E	I	Bus Read	E	X
P1 read A[N/2]	Bus Read	I	E	Bus Read	X	E
P0 reads B[0]	Bus Read	E	I	Bus Read	E	X
P1 read B[N/2]	Bus Read	I	E	Bus Read	X	E
P0 writes A[0]		M	I		M	X
P1 writes A[N/2]		I	M		X	M
P0 reads A[1]		M	I		M	X
P1 read A[N/2+1]		I	M		X	M
P0 reads B[1]		E	I		E	X
P1 read B[N/2+1]		I	E		X	E
P0 writes A[1]		M	I		M	X
P1 writes A[N/2+1]		I	M		X	M

Number of bus transactions for MESI (all iterations) =

4 bus transactions (to process 8 elements) × N/8 = 4 N / 8 = N/2

Number of bus transactions for Dragon (all iterations) =

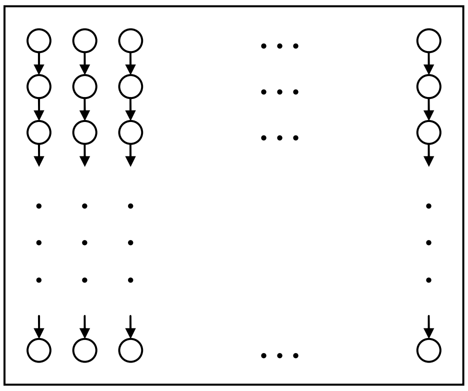
4 bus transactions (to process 8 elements) × N/8 = 4 N / 8 = N/2

No difference between MESI and Dragon in this case

Q3. (25 pts) Consider the following nested loops, where **A** and **B** are 2-dimensional arrays with 64×64 elements.

```
for (i=1; i<64; i++)
  for (j=0; j<64; j++)
    A[i][j] = B[i][j] + A[i-1][j] * B[i-1][j];
```

(a) (5 pts) Draw matrix A showing the data dependences among its elements and indicate whether the outer or inner loop can be vectorized.



Each row in matrix A depends on the previous one.

Inner loop can be vectorized

(b) (10 pts) Translate the above nested loops to VMIPS assembly code (see Figure G.3 for the VMIPS vector instructions). The VMIPS vector registers have 64 elements, which match the number of elements along each dimension of matrices **A** and **B**. Here is a sample of VMIPS instructions that you might find useful:

ADDV.D	**V3, V1, V2**	**# V3 = V1+V2 (64-element vector registers)**
MULV.D	**V3, V1, V2**	**# V3 = V1*V2 (64-element vector registers)**
LV	**V1, R1**	**# Load V1 from memory starting at address R1**
SV	**R1, V1**	**# Store V1 into memory starting at address R1**

```
    # Each row consists of 64 elements * 8 bytes = 512 bytes
    # Initially R1 & R2 contain addresses of Matrices A & B
    LV      V1, R1        # load vector A[0][0:63]
    LV      V2, R2        # load vector B[0][0:63]
    LI      R3, 63        # 63 iterations in the outer loop
loop:
    ADDI    R1, R1, 512   # R1 = pointer to next row in A
    ADDI    R2, R2, 512   # R2 = pointer to next row in B
    MULV.D  V1, V1, V2    # V1 = A[i-1][0:63] * B[i-1][0:63]
    LV      V2, R2        # load vector B[i][0:63]
    ADDV.D  V1, V1, V2    # V1 = V1 + B[i][0:63]
    SV      R1, V1        # store vector A[i][0:63]
    ADDI    R3, R3, -1    # R3 = loop counter
    BNE     R3, 0, loop   # loop back if R3 != 0
```

(c) (10 pts) Estimate the execution time of the above nested loops on a VMIPS vector processor. The VMIPS is a single-issue processor that can issue one instruction per clock cycle and has only one lane. It has one FP add/subtract unit, one FP multiply unit, and one vector load/store unit. The functional units are deeply pipelined and have a startup latency overhead equal to 6 cycles for the FP add/subtract unit, 5 cycles for the FP multiply unit, and 20 cycles for the vector load/store unit. There is also an integer ALU for executing scalar instructions at the rate of 1 cycle per scalar instruction.

If **no chaining** is provided, how many cycles does it take to compute all elements of matrix **A**?

Solutions will vary depending on the code in part (b)

Cycles outside loop = 168 cycles because there is only one load/store unit

2 LV = 2 × (20 (startup cycles) + 64 (load cycles)) = 168 cycles

Cycles per iteration = 84 + 70 + 84 = 238 cycles

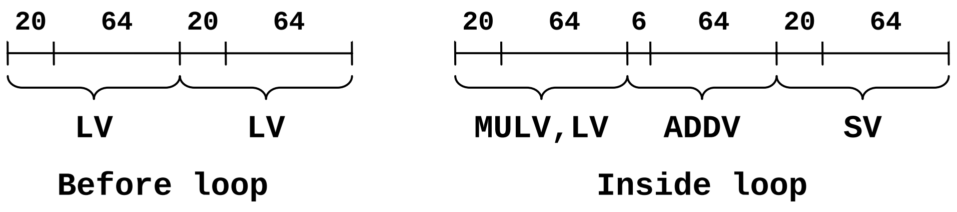
MULV + LV are placed in the same convoy = 20 + 64 = 84 cycles

ADDV = 6 + 64 = 70 cycles

SV = 20 + 64 = 84 cycles

Scalar instruction execution is overlapped with vector instruction execution

Total cycles = 168 + 63 * 238 = 15,162 cycles



If **chaining** is now supported, how many cycles does it take to compute all elements of matrix **A**?

Cycles outside loop = 168 cycles (same as above)

Inside loop:

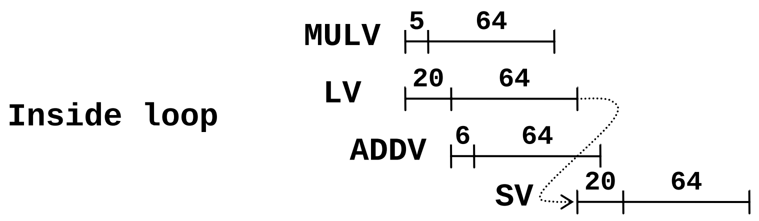
LV and ADDV can be chained

However LV and SV cannot be chained (only one Load/Store unit)

Cycles per iteration = 20 + 64 + 20 + 64 = 168 cycles

Total cycles = 168 + 63 * 168 = 10,752 cycles

Scalar instructions are overlapped with vector instructions (as above)



Q4. (25 pts) Consider the following pseudocode describing sequential Gaussian elimination:

```
procedure Eliminate(A)
begin
  for k = 0 to n – 1 do
  begin
     for j = k+1 to n – 1 do
        A[k][j] = A[k][j] / A[k][k];
     end for
     A[k][k] = 1;
     for i = k+1 to n – 1 do
        for j = k+1 to n – 1 do
           A[i][j] = A[i][j] – A[i][k] * A[k][j];
        end for
        A[i][k] = 0;
     end for
  end for
end procedure
```

Assuming a decomposition into rows and an assignment into blocks of adjacent rows, **write a shared address space parallel version** using *LOCK* and *BARRIER* primitives for synchronization. Assume the existence of *P* processes executing *Eliminate*(*A*) in parallel. Indicate which variables are shared and which ones are private to each process.

Answer:

The Array *A*, the number of processes *P*, and the barrier *bar* are shared by all processes. The local variables declared inside the procedure *Eliminate* are private in each process. The *pid* variable is assumed to be unique in each process and assumes the values 0 through *P* – 1.

```
procedure Eliminate(A)
begin
  pid = getrank();            // unique number = 0 to P-1
  for k = 0 to n – 1 do
  begin
     if (pid == 0) then       // done only by first process
        for j = k+1 to n – 1 do
           A[k][j] = A[k][j] / A[k][k];
        end for
        A[k][k] = 1;
     end if
     BARRIER(bar, P);         // All processes wait for P0
     rows = (n-k-1)/P;        // rows assigned per processor
     min = k+1 + rows * pid
     max = min + rows – 1;
     if (pid == P-1) max = n-1; end if
     for i = min to max do    // rows are partitioned here
        for j = k+1 to n – 1 do
           A[i][j] = A[i][j] – A[i][k] * A[k][j];
        end for
        A[i][k] = 0;
     end for
     BARRIER(bar, P);         // All processes wait here
  end for
end procedure
```