CS 15-740, Fall 1994

Final Exam

December 9, 1993

Instructions:

1. Make sure that your exam is not missing any sheets.
2. You must turn in your solutions by 8:00 pm to WeH 7128.
3. A well-prepared student should be able to complete this exam in 3 hours. The exam has 180 points total, indicating a rough breakdown of 1 minute per point.
4. Do not overkill the problems. Clear and concise answers are sufficient, showing that you understand the key points of the question.
5. You may use any material you wish, including documents from the library, the WWW, etc.
6. No collaboration of any form is allowed. Do not discuss the contents of this exam with anyone other than the instructor, until after the 8:00 pm deadline.
7. Prepare your solution *as a separate document*. Make sure that the problems are ordered as in the exam. Make sure your document is clearly legible (typewritten preferred). You do not need to do fancy text formatting or figure drawing.
8. If you feel certain necessary details are missing from the problem statement, you may make reasonable assumptions, but state them clearly.
9. If you have any questions during the exam, you may contact the instructor as follows:
   - In person in room WeH 7128. I will either be there or have a sign posted on the door with a forwarding address. I will be out of my office from 12:00 to 1:30 and from 5:30 to 7:00 pm.
   - By phone at 268-8821.
   - By electronic mail (Randy.Bryant@cs.cmu.edu).
10. During the day, any clarifications will be made via the "Finals" Web page, accessible from the course home page.

1. Performance of Pipelines and Caches

The following series of problems concern executing code on a MIPS R2000 processor. The following facts are known about the processor:

- The architecture is very similar to the pipelined version of DLX presented in Chapter 6 of Hennessy & Patterson.
- There is a single branch delay slot.
- The floating point performance is as described in Figure 6.29 (p. 289).
- The assembly code command `l.d` (load double precision), is expanded by the assembler into two loads of 4 bytes each.
- The page size is 4KB.
- All data in the following problems are aligned to begin on page boundaries.
- There is a 64KB direct mapped data cache, and a separate instruction cache.

The following 3 functions compute the "inner product" of two vectors X and Y. Each vector consists of 512 double precision floating point numbers.

```
/* No interleaving */
double inner_prod1(double x[], double y[])
{
  int i;
  double result = 0.0;
  for (i = 0; i < 512; i++) {
    double xt0 = x[i];
    result += xt0 * y[i];
  }
  return result;
}

/* Two way interleaving */
double inner_prod2(double x[], double y[])
{
  int i;
  double result = 0.0;
  for (i = 0; i < 512; i+=2) {
    double xt0 = x[i];
    double xt1 = x[i+1];;
    result += xt0 * y[i];
    result += xt1 * y[i+1];
  }
  return result;
}
```

```
/* Four way interleaving */
double inner_prod4(double x[], double y[])
{
  double p;
  int i;
  double result = 0.0;
  for (i = 0; i < 512; i+=4) {
    double xt0 = x[i];
    double xt1 = x[i+1];
    double xt2 = x[i+2];
    double xt3 = x[i+3];;
    result += xt0 * y[i];
    result += xt1 * y[i+1];
    result += xt2 * y[i+2];
    result += xt3 * y[i+3];
  }
  return result;
}
```

Running GCC with optimization level 2 generates the following assembly code for the loop of `inner_prod1`:

```
$L5:
      l.d      $f0,0($4)
      l.d      $f2,0($5)
      mul.d    $f0,$f0,$f2
      addu     $5,$5,8
      addu     $4,$4,8
      addu     $3,$3,1
      slt      $2,$3,512
      bne      $2,$0,$L5
      add.d    $f4,$f4,$f0
```

Problem 1. (10 points):

Under ideal conditions, `inner_prod1` requires 12 clock cycles per vector element. Give a detailed accounting for this performance.

Problem 2. (14 points):

When the performance is measured by repeatedly executing the function on the same vectors, the performance normally matches that of the ideal case. For some data allocations, however, the performance is significantly worse. The following table shows the measurements for the three functions. All times are measured in cycles per vector element. [Note that the assembly code was manually modified to undo the rearranging of loads by the compiler.]

Interleaving	Normal	Anomalous
1	12	68
2	10	38
4	9	37

A. Why does the "normal" performance improve as the degree of interleaving is increased?

B. What is the source of the anomalous behavior?

C. What can you infer about the cache structure and performance from these measurements?

Problem 3. (18 points):

Suppose we wish to compute the product of a 512×512 matrix times a 512 element vector, by computing the inner product (using `inner_prod1`) of the vector times each row of the matrix. Assume initially that neither the matrix nor the vector is loaded into the cache.

A. How many memory accesses to the matrix or vector elements are made in computing this product?

B. Assuming there were no cache collisions, how many of these accesses would cause cache misses?

C. Including the effects of collisions, how many of the accesses would you expect to cause a cache miss?

D. What is the effect of the cache on the overall performance of the computation?

2. Vector Machines

This next series of problems concern techniques for performing a search in an ordered set of data on a DLXV vector machine, having the following characteristics.

- The vector units are as listed in Figure 7.2 (p. 355) of Hennessy & Patterson.
- Its instruction set includes those of Figure 7.3, as well as the following, implemented by the integer add unit:

Instruction	Operands	Function
`ADDIV`	`V1, V2, V3`	Add integer vectors
`ADDISV`	`V1, R1, V2`	Add integer scalar to vector

- Vectors are 64 words long.
- The load pipe has a 20 cycle startup, and (unless there's a bank conflict) can then deliver one word per cycle.
- Other pipeline instructions have a 6 cycle startup and can then deliver one word per cycle.
- The vector operations can chain, but a vector register can be the source for only one chaining operation at a time.
- A stall due to a structural hazard can be released immediately after the last element from the conflicting operation has been deposited into the pipeline, i.e., it is not necessary to wait until the pipeline drains.
- For the purpose of these problems, assume all scalar instructions take 0 time (i.e., their executions overlap the memory and vector instructions).
- The memory is divided into 256 banks. Each bank has a cycle time of 4.

The search problem is defined as follows:

- The data are stored as an array (of floats) `X` of 2^{16} elements, sorted in ascending order. The keys are stored as an array (of floats) `Y` of 64 elements (unsorted).
- Given value `Y[j]`, we are to find index `indx[j]` such that `X[indx[j]]` $\leq$ `Y[j]` $<$ `X[indx[j]+1]`.
- We may assume that `X[0]` $\leq$ `Y[j]` $<$ `X[(1<<16) - 1]` for all values of `j`.
- The keys and the data follow the same probability distribution.

The usual way to solve this would be by binary search, using code of the following form:

```
for (j = 0; j < 64; j++) {
 indx[j] = 0;
 for (int offset = 1<<15; offset > 0; offset = offset>>1)
 if (Y[j] >= X[indx[j]+offset])
      indx[j] += offset;
}
```

Problem 4. (15 points):

Describe how you would write this code to vectorize over the keys, i.e., performing all 64 searches simultaneously. You need not give detailed assembly code, but you should make clear what vector instructions are being used.

Problem 5. (19 points):

Ignoring problems due to bank conflicts, determine the time required by each iteration of the binary search. Show a timing diagram illustrating the utilization of the vector units for each iteration, and how the different iterations would align. What are the factors constraining the performance of the computation?

Problem 6. (14 points):

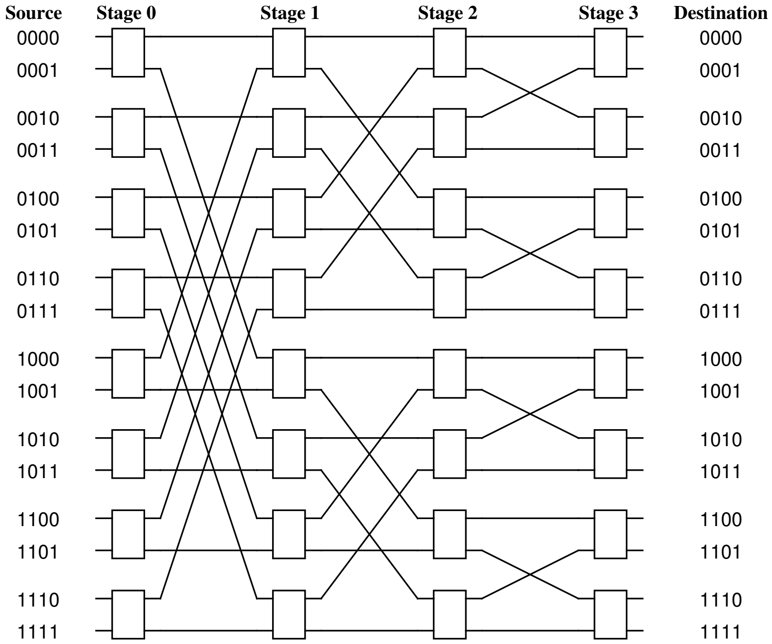
Now consider the effects of bank conflicts.

A. Estimate the performance penalty due to bank conflicts. You need not give a detailed probabilistic analysis. Make simplified estimates of the major sources of conflict.

B. How large would this penalty be relative to the overall running time of the program?

C. Suggest a method to reduce the major bank conflicts.

3. Routing in Butterfly Networks



Consider an r-dimensional butterfly network, such as the one illustrated above for $r = 4$. Such a network can route messages from 2^r sources (on the left) to 2^r destinations (on the right). The network has r stages, each consisting of 2^{r-1} switches. Each switch can route a message from either input port to either output port. [Note that this formulation of butterfly networks differs slightly from that of Leighton. He considers the left and right columns to be the sources and destinations, respectively. He would classify the above network as having dimension 3, routing from 8 sources to 8 destinations.]

For each stage, we will number the 2^r input ports (two per switch) and the 2^r output ports from top to bottom, just is as shown above for the source and destination addresses.

Problem 7. (20 points):

The connections between stage i and $i + 1$ can be described in terms of a permutation $\pi_i(y)$, giving the input port of stage $i + 1$ connected to output port y of stage i.

A. Define the connection permutation between each pair of stages in a butterfly network. This can best be done in terms of the binary representation of y.

B. In the standard algorithm for routing messages in a butterfly, a message arriving at a stage i switch is routed either up or down, according to bit $r - i - 1$ (with bit 0 being the LSB) of the destination address. Justify this algorithm in light of your permutation definitions.

Problem 8. (15 points):

Suppose we want to route messages through a *reverse butterfly*, i.e., as shown above, but with the sources on the right and the destinations on the left.

A. Devise a routing algorithm.

B. Justify your reverse routing algorithm.

4. Synchronization with Fetch and Add

In the mid-80's researchers on the NYU Ultracomputer project advocated an atomic *fetch-and-add* instruction, as a synchronization primitive for shared memory parallel processors, rather than the more traditional "test-and-set" or "atomic swap."

Assume we add this to the DLX instruction set as follows:

`FCHADD R1,30(R2)` Fetch and add word, atomic

$$\begin{aligned} \textit{temp} &\leftarrow_{32} \mathrm{M[30{+}R2]} \\ \mathrm{M[30{+}R2]} &\leftarrow_{32} \textit{temp} + \mathrm{R1} \\ \mathrm{R1} &\leftarrow_{32} \textit{temp} \end{aligned}$$

With atomic execution, we are guaranteed that the memory location cannot be altered between the read and write of the instruction.

Problem 9. (10 points):

Consider the MDLX microcoded implementation of DLX, as used in Assignment 2 and documented in the file:

`/afs/cs.cmu.edu/academic/class/15740-f94/public/doc/mdlx.ps`

A. Show a series of state templates implementing the fetch-and-add instruction.

B. What properties do you require of the memory interface to guarantee atomicity?

C. How would you handle a page fault event for the data address?

Problem 10. (18 points):

Assume that we want to implement a locking mechanism, guaranteeing exclusive access by a single process to a "critical section." Sketch code for routines `acquire` and `release` such that acquiring a lock requires a single fetch-and-add. Furthermore, your mechanism should be guaranteed to be fair, as long as the implementation of fetch-and-add is fair.

Problem 11. (12 points):

Consider the bus-based cache protocol described in Hennessy and Patterson (pp. 489–474), and presented in more detail in Assignment 5.

A. What features must be added to the system to support fetch-and-add? Describe in general terms how the protocol would be modified.

B. Describe the bus traffic generated due to synchronization under the following scenario: A number of processes are contending for a shared resource on a continual basis. This resource is guarded by your locking mechanism from the previous problem. When a process acquires exclusive access to the resource, it will typically hold it for several hundred cycles, release it, and then attempt to reacquire it several hundred cycles later.

Problem 12. (15 points):

The developers of the Ultracomputer envisioned a system with a number of processors connected to a number of memory banks via a routing network. They proposed providing hardware support for the fetch-and-add instruction, such that instructions operating on the same memory location could be "combined" at intermediate points in the network, thereby avoiding a bottleneck at the memory bank.

A. Describe in general terms how the switches in the routing network must be embellished to provide this support. How would the combining be implemented?

B. How does this compare to complexity of the hardware support for scan operations provided by the Connection Machine CM-5, as described by Leiserson?