Full Name:_______________________________________

Email:_______________________________________

CS 15-740, Fall 1996

Final Exam

Dec. 11, 1996

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and email address *on every indicated sheet of the exam.*
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 168(!) points.
- If you feel certain details are missing from the problem statement, you may make reasonable assumption, but state them clearly.
- This exam is OPEN BOOK. You may use any books or notes you like. You're even allowed to leave the room and do the exam elsewhere, although any form of collaboration is prohibited.

Do not write below this line

Pipe	Cache	Consistency	Butterfly	NESL	Vector
Total: 168: _____					

1. Pipeline Processor Implementation

Consider a pipeline architecture for a RISC processor similar to DLX or MIPS, but without any "delay slots" in the instruction set architecture. Assume this processor is implemented with a 4-stage pipeline as follows:

Stage	Instruction Type		
	ALU	Load/Store	Branch
Fetch	Fetch Instr.	Fetch Instr.	Fetch Instr.
Decode	Read Reg.	Read Reg. Compute EA	Read Reg. Compute Cond. Compute target
Execute	ALU function	Read/Write mem.	If taken, update PC
Write	Update reg.	Update reg.	

This pipeline differs from the usual 5-stage pipeline in that the memory operation takes place in the execute stage. This necessitates computing the effective address (EA) during the decode stage. Also, the branch condition and target is computed in decode, rather than execue.

Assume the following about the implementation:

- On each cycle, the register file performs the read for the decode stage *before* the write for the writeback stage.
- Full bypassing is provided for handling data hazards. Some stalling may still be required, though.
- Control hazards are handled by stalling.

Problem 1. (24 points):

Fill in the following table concerning the pipeline behavior for different instruction combinations. Enter in the column "Stall Count" the number of cycles that the pipeline would need to stall. Enter in the third and fourth columns what form of bypassing would occur (or "NA" if no bypassing is required), where "From" indicates the just-completed stage that provides the datum, and "To" indicates the stage that uses it. The notation "ALU $\rightarrow$ NOP ALU" means two ALU instructions separated by some unrelated intervening instruction (e.g., a NOP), while "ALU $\rightarrow$ NOP^2 ALU" means two ALU instructions separated by two unrelated intervening instructions.

Hazard Class	Stall Count	Bypass From	Bypass To
ALU $\rightarrow$ ALU			
Load $\rightarrow$ ALU			
Load $\rightarrow$ Store			
ALU $\rightarrow$ Load Data			
ALU $\rightarrow$ Store Address			
ALU $\rightarrow$ Branch Data			
ALU $\rightarrow$ NOP ALU			
ALU $\rightarrow$ NOP^2 ALU			

Problem 2. (6 points):

How many cycles would the processor need to stall for a taken branch? For an untaken one? Explain how these cases would progress through the pipeline.

2. Cache Performance

Consider the following code for summing the elements of an array of length n a total of t times:

```
int sum(int *a, int n, int t)
{
  int i;
  int sum = 0;
  while (t-- > 0)
    for (i = 0; i < n; i++)
      sum += a[i];
  return sum;
}
```

Assume this code is executed on a machine with separate instruction and data caches, and where the cache holds m words. The page size of the machine is p words, and you may assume that both n and m are multiples of p. Assume also that t is high enough that the cost of initially loading the cache can be ignored.

Problem 3. (9 points):

Suppose the cache is direct mapped and virtually indexed. Derive a formula describing the hit rate.

Problem 4. (9 points):

Suppose the cache is virtually indexed, but also that it is 2-way set associative. Cache lines are replaced within a set according to an LRU policy. Derive a formula describing the hit rate.

Problem 5. (12 points):

Suppose the cache is physically indexed and direct mapped. Assume also that there is essentially no correlation between the bit patterns defining the virtual and physical page numbers of the pages. Derive a formula estimating the expected hit rate. [**Big Hint**: If one were to distribute a set of k balls among a set of b buckets, the expected number of buckets containing exactly one ball would be approximately $k \cdot e^{-k/b}$.]

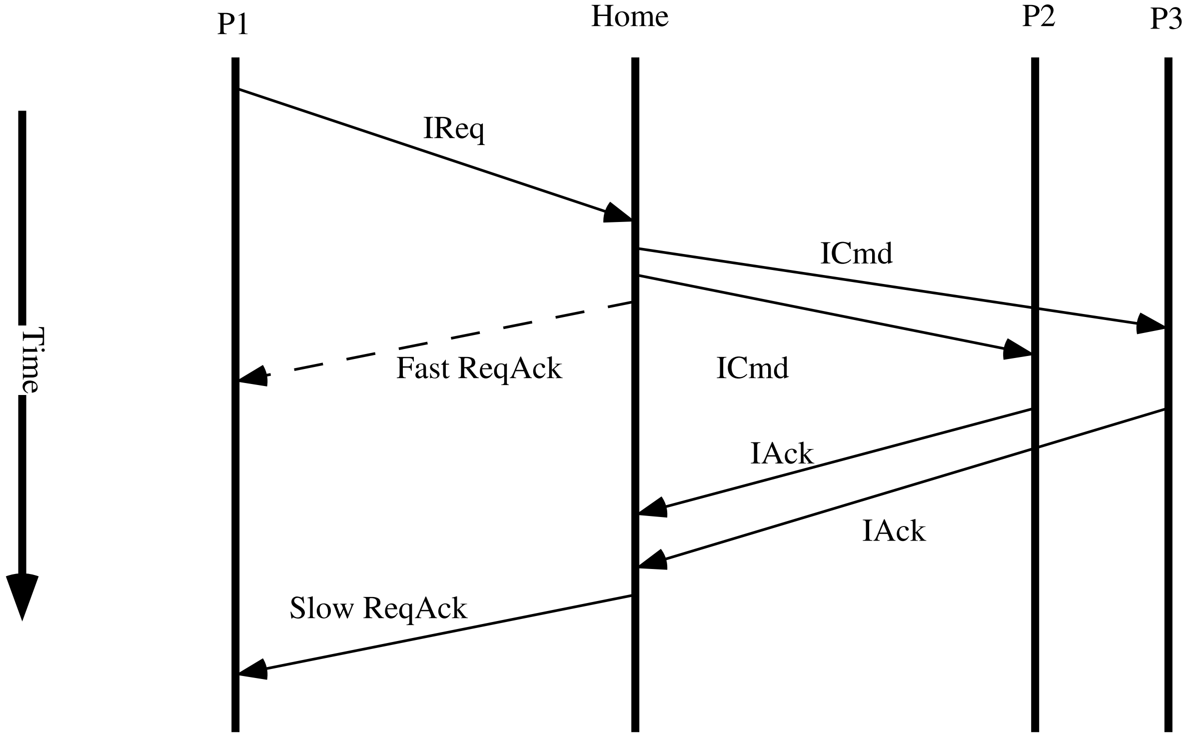

Figure 1: Transactions for P1 to gain exclusive access to previously-shared block

3. Directory-Based Cache-Coherence

Consider a distributed shared memory system using a directory-based cache coherency scheme. Each block has a “home,” i.e., a cache controller responsible for managing its status. Suppose three processors P1, P2, and P3 have shared copies of some block, and P1 wants to gain exclusive access in order to write to a word in this block.

A series of transactions to obtain this privilege is illustrated in Figure 1. In this figure, time moves downward, with each vertical line denoting the actions of an individual agent, and the arrows denoting communications among them. The communications are:

IReq P1 sends a invalidation request to the home.

ICmd The home sends invalidation commands to the other processors having copies of the block

IAck The other processors acknowledge the invalidation command

ReqAck The home acknowledges the original request.

In this diagram, two versions of ReqAck are diagrammed: “Fast” where the request is acknowledged immediately, and “Slow” where the home waits until it is sure all other copies have been invalidated.

Problem 6. (5 points):

Suppose that P1 begins writing to its block before it receives any acknowlegement. Describe a scenario in which this could have disasterous consequences, where two processors accessing independent memory words could interfere.

Problem 7. (5 points):

Suppose we use the "Fast" version of ReqAck, and that P1 can begin writing to its block as soon as it receives this acknowledgement. Describe a scenario in which a concurrent program (e.g., the one shown at the bottom of H&P page 708) could violate sequential consistency.

Problem 8. (10 points):

The MIPS instruction set includes an instruction `SYNC` which, when executed, guarantees that all preceding memory operations by the program are completed before any successive ones are begun. Describe how such an instruction could be implmented. State what resources the processor would require, how they would be managed, and and how it would behave with respect to the transactions diagrammed in Figure 1.

4. Implementing Prefix Computations on Butterfly Networks

In Assignment 5 we proved that two back-to-back butterflies form a rearrangeable network, as do a series of four butterflies. Butterflies have many other nice features as well. For example, the Fast Fourier Transform algorithm can be implemented directly on a butterfly network. In this problem, we are going to design a simple algorithm for performing prefix computations on butterfly networks.

To refresh your memory, a prefix computation takes an input sequence $a_0, a_1, \ldots, a_{n-1}$, and a binary associative operator $\oplus$, and produces an output sequence $b_0, b_1, \ldots, b_{n-1}$, where $b_i = a_0 \oplus a_1 \oplus \cdots \oplus a_i$. (An operator $\oplus$ is associative if $(c \oplus d) \oplus e = c \oplus (d \oplus e)$, for any c, d, and e.) As an example, when $\oplus$ is integer addition, b_i is the sum of the numbers a_0 through a_i. Similarly, when $\oplus$ is the min operator, b_i is the minimum of the values a_0 through a_i.

For the purposes of this problem, the butterfly network is the one shown below.

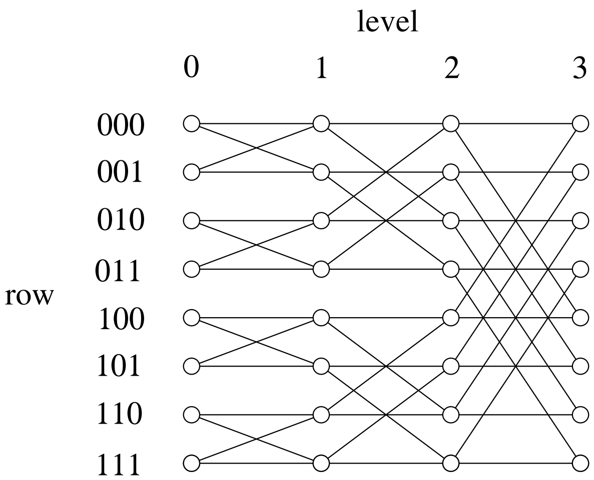

Figure 2: An 8-input butterfly network.

Suppose that the elements of a sequence $a_0, \ldots, a_{n-1}$ enter the butterfly at the nodes in level 0 labeled with the binary numbers 0 through $n - 1$.

Problem 9. (10 points):

Design an algorithm that produces the sum $a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}$ at each node in the rightmost level. (*Note:* this is easier than performing a prefix computation.) In your algorithm data should flow through the butterfly from left to right only. You may assume that every node in the butterfly is capable of performing an $\oplus$ machine, and can store a small number of values. Explain the rule used at each node.

Problem 10. (5 points):

Prove (by induction) that your algorithm computes, at each node u anywhere in the butterfly, the sum (using $\oplus$) of all of the input values that can reach node u. The base case is easy: the sum for each node in the first level is just the input element that starts at that node. For the inductive step, note that if we remove the last level of edges and the last level of nodes from the butterfly, we are left with two $n/2$-input subbutterflies, one consisting of the top $n/2$ rows, and the other consisting of the bottom $n/2$ rows.

Problem 11. (10 points):

Using the algorithm that you have just designed, we are now going to design an algorithm that produces the outputs $b_0, \ldots, b_{n-1}$ of the prefix computation at the corresponding nodes in the last level of the butterfly. Here is a major hint: your algorithm should send two values across each edge of the butterfly. One of these should be exactly the same as the value sent by the previous algorithm. Here is another major hint: suppose that we were to remove the nodes in levels $l + 1$ through $\log n$ from the butterfly network, along with all edges incident on those nodes. What remains is a collection of disjoint subbutterflies spanning levels 0 through l. Your algorithm should perform a prefix computation for each subbutterfly. Give the rule for computing the values sent along each edge.

Problem 12. (5 points):

Prove (by induction) that your algorithm is correct.

5. Implementing Nested Parallelism in NESL

The following NESL function implements the Quicksort algorithm. In order to simplify this problem we have assumed that all of the elements in the sequence to be sorted have distinct values.

```
function quicksort(a) =
if (#a < 2) then a
else
  let pivot   = a[rand(#a)];
      lesser  = {e in a| e < pivot};
      greater = {e in a| e >= pivot};
      result  = {quicksort(v): v in [lesser,greater]};
  in result[0] ++ result[1];
```

This function exhibits *nested* parallelism because the quicksort function, which is itself parallel, is applied recursively to all of the elements of the sequence `[lesser,greater]` (which happens to have only two elements). Note that in NESL the only parallel operation is to apply a function to all elements of a sequence simultaneously. The `quicksort` function has expected depth $O(\log n)$ and expected work $O(n \log n)$, where n is the length of the sequence `a`.

Without nested parallelism, we might be forced to write the function as follows.

```
function quicksort2(a) =
if (#a < 2) then a
else
  let pivot   = a[rand(#a)];
      lesser  = {e in a| e < pivot};
      greater = {e in a| e >= pivot};
      result0 = quicksort2(lesser);
      result1 = quicksort2(greater);
  in result0 ++ result1;
```

Problem 13. (8 points):

Suppose that function `quicksort2` is called on an input sequence with n elements. Using big-O notation, what is the expected depth and work of the function?

In order to evaluate an expression with nested parallelism, NESL first flattens the nested sequence to form a single un-nested sequence, and then applies a function to the un-nested sequence. Suppose that we manually remove the nested parallelism as follows.

```
function quicksort3(a) =
if (#a < 2) then a
else
  let pivot   = a[rand(#a)];
      lesser  = {e in a| e < pivot};
      greater = {e in a| e >= pivot};
      in quicksort3(flatten([lesser,greater]));
```

Problem 14. (4 points):

What is wrong with this function?

NESL implements nested parallelism by breaking the flattened sequence into segments, and then performing segmented operations on the sequence. These segments are the same as those defined in Assignment 3. In the next part of this problem, we are going to implement a segmented prefix computation in NESL. A segmented addition operator is specified by the table below.

$\oplus$	d	$/d$
c	$c + d$	$/d$
$/c$	$/c + d$	$/d$

The following function implements this table. Here each element is represented as an ordered pair. The first component of the pair indicates whether a slash is present, and the second holds the data. For example, $/c$ is represented as `(t,c)` and d is represented as `(f,d)`, where `t` and `f` are NESL's built-in boolean values.

```
function segsum(((slash_c,c),(slash_d,d))) =
let
  slash = if slash_c or slash_d then t else f;
  sum = if slash_d then d else c+d;
  in (slash,sum);
```

NESL calls parallel prefix computations "scans". In Assignment 3, we gave a recursive algorithm for performing a parallel prefix computation. For your convenience, that algorithm is repeated here:

1. Compute the sums of pairs of consecutive elements in a, i.e., compute $g_i = a_{2i} \oplus a_{2i+1}$, for $0 \leq i < n/2$ (let's assume that n is even).
2. Recursively perform a prefix computation on the sequence $g_0, \ldots, g_{(n/2)-1}$. Let $h_0, \ldots, h_{(n/2)-1}$ denote the output sequence produced by this computation.
3. Compute the final output values $b_0, \ldots, b_{n-1}$ as follows:

$$b_i = \begin{cases} a_0 & i = 0 \\ h_{(i/2)-1} \oplus a_i & i \text{ even}, i > 0 \\ h_{(i-1)/2} & i \text{ odd} \end{cases}$$

Problem 15. (10 points):

Your task is to write a NESL function that implements this algorithm. To help you get started, we have provided some of the code. (In case you are curious, NESL doesn't perform segmented prefix computations this way. For better efficiency, they are implemented directly in the underlying run-time system.)

```
function segmented_plus_scan(a) =
if (#a == 2) then ([a[0]]++[segsum((a[0],a[1]))])
else
  let
    g = ____________________________________________________;

    h = ____________________________________________________;

    b = {if (i == 0) then ___________________________________________

        else if ((i/2)*2) == i then _______________________________

        else _____________________________________________________

        : i in [0:#a]};

    in b;
```

Problem 16. (5 points):

What is the work and depth of your function?

With some effort, it is possible to write a parallel version of the quicksort function that uses segmented scans instead of nested parallelism. But it's too difficult for an exam question. Instead, we'll write a function that scans all of the rows in a dense matrix. The code that uses nested parallelism is shown below. The input to this function is a sequence of sequences. For example, `[[1,2],[3,4]]` represents a 2×2 matrix whose first row consists of the elements 1 and 2 and whose second row consists of the elements 3 and 4.

```
function scan_rows(a) =
{plus_scan(r) : r in a};
```

Problem 17. (10 points):

Your task is to write a new version of this function that uses segmented scans (i.e., the function `segmented_plus_scan`) instead of nested parallelism. To help you out, we have provided some of the code. (Note: NESL defines the tt plus_scan function a little differently than we have in this problem. Don't worry about this.)

```
function new_scan_rows(a) =
let
  % replace each matrix value by a (flag,value) pair %
  b = {{if (i == 0)
          then (t,val)
          else (f,val) : val in r; i in [0:#r]} : r in a};


  c = _______________________________________________;


  d = _______________________________________________;

  % unflatten the sequence and remove flags %
  e = partition(d,dist(#a[0],#a));
  g = {{val : (flag,val) in r} : r in e};
  in g;
```

6. Implementing Scans on a Vector Machine

The following C function performs a scan on an array a of floating-point values.

```
#include<stdlib.h>

double *scan(double *a,int n){
  int i;
  double sum = 0.0;
  double *b = (double *) malloc(sizeof(double)*n);
  for (i = 0; i < n; i++){
    sum = sum+a[i];
    b[i] = sum;
  }
  return b;
}
```

Problem 18. (8 points):

What makes this function difficult to vectorize?

Our goal will be to design a vectorized algorithm for performing a scan on a vector machine with 16 vector registers, each containing 64 words. To make things simple, let us assume that our array has length $64 \cdot 8$. (This will give us a few scratch registers.) Suppose that our machine has the following vector operations:

1. Sum the words in a vector register.
2. Add a scalar constant to each word in a vector register.
3. Add two vector registers.

Here is the algorithm.

1. View the $(64 \cdot 8)$-word array as if it were a 64×8 matrix (i.e., 64 rows and 8 columns) laid out in row-major order. Put each column into a vector register.
2. Sum the elements in each row. Let $r_0, \ldots, r_{63}$ denote these sums.
3. Perform a sequential scan on $r_0, \ldots, r_{63}$. Denote the output of this scan $s_0, \ldots, s_{63}$.
4. Perform a scan on each row.
5. Add s_i to each element in row $i + 1$.

Problem 19. (8 points):

Explain how Steps 2, 4, and 5 are vectorized.

Problem 20. (5 points):

Explain how Steps 2 and 4 can be merged to form a single step.