Name _____

Computer Architecture
EE 4720
Final Examination

11 May 2001, 15:00–17:00 CDT

Problem 1	_____	(25 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(25 pts)
Problem 4	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: The *reference count* of a value stored in a register is the number of times the value is referenced (read) by instructions before being overwritten. In the example below the reference count of the value written by the `add` is zero because it is overwritten before being read. The reference count of the value written by the `lw` is two.

```
add r1, r2, r3   ! A first value for r1 is defined (written).
lw  r1, 0(r4)    ! A second value for r1 is written, the first one is never used.
sub r5, r5, r1   ! The second value is referenced (read).
xor r6, r1, r7   ! The second one is referenced again.
or  r1, r0, r0   ! A third value for r1 is defined.
```

Three new instructions are to be added to DLX to obtain reference count information. The instructions refer to two registers, `rc.z` and `rc.nz`. Register `rc.z` holds the number of values written to `r1` with a zero reference count and `rc.nz` holds the number of values written to `r1` with a non-zero reference count. The counts are updated on either the first reference or when a new value is defined, whichever is earlier.

Instruction `rc.reset` sets both of these registers to zero. Instruction `movstoi RD, rc.z` moves the contents of `rc.z` to a general-purpose register (shown as `RD`), `movstoi RD, rc.nz` moves the contents of `rc.nz` to a general-purpose register.

The instructions are used in the code below.

```
rc.reset           ! rc.z -> 0,  rc.nz -> 0
add r1, r2, r3
lw  r1, 0(r4)      ! rc.z -> 1
sub r5, r5, r1     ! rc.nz -> 1
xor r6, r1, r7
and r1, r0, r0
or  r1, r0, r0     ! rc.z -> 2
slli r1, r1, #1    ! rc.nz -> 2
movstoi r8, rc.z   ! r8 -> 2
movstoi r9, rc.nz  ! r9 -> 2
add r10, r8, r9    ! Total number of writes to r1.
```

The new instructions should work for `r1` and no other register. (25 pts)

(*a*) For each new instruction below cross out the instruction type that could not reasonably be used to code it. Circle the type that would best be used to code it.

☐ `rc.reset` Possible types: Type R, Type I, Type J.

☐ `movstoi RD,rc.z` Possible types: Type R, Type I, Type J.

(*b*) As the alert test taker has noticed, there is already a `movstoi` instruction, used to move a special-purpose register (such as the processor status word, not covered much in class) to a general-purpose register.

☐ The new `movstoi RD,rc.z` instruction can be coded as a new instruction (with its own opcode) or as a new use of `movstoi`. That would depend on how the existing `movstoi` is defined. Explain that. *Hint: Suppose there was only one special register in DLX.*

Problem continued on next page.

(*c*) Modify the pipeline below to implement the new instructions. The code on the previous page should execute properly.

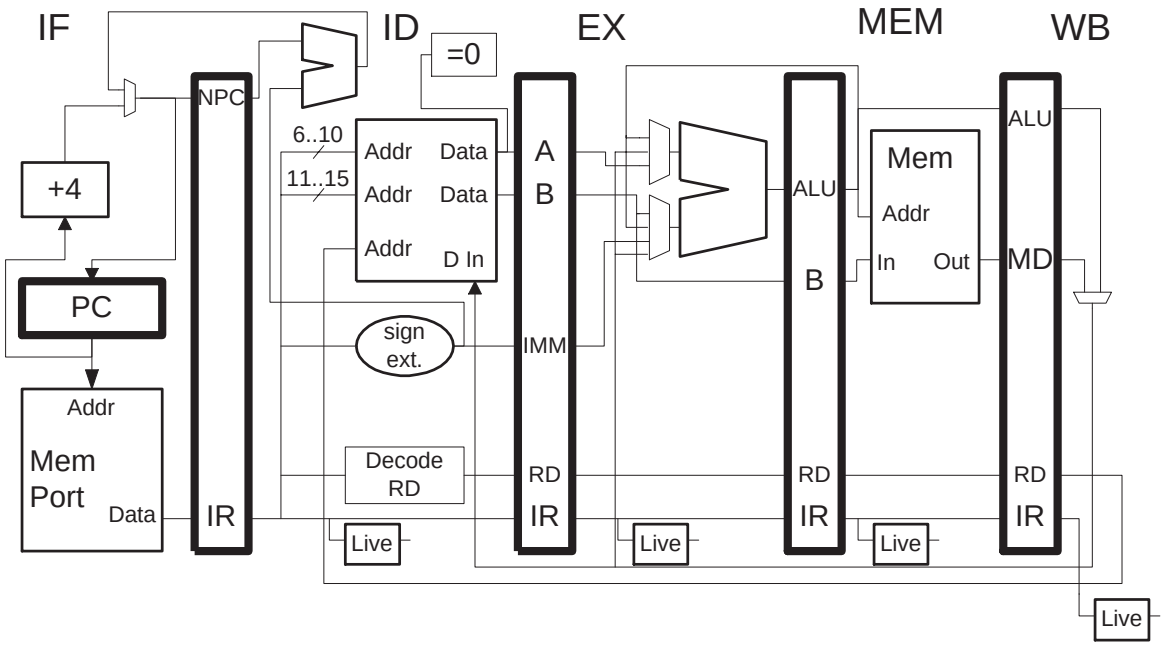
Assume that all Type-R instructions have two source operands and all Type-I instructions have one source operand. Use `is Type R` and `is Type I` to recognize these instruction types.

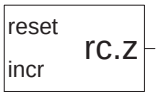
Assume that Decode RD in the diagram can recognize the new instructions

☐ The counts should not include squashed instructions.

☐ Show control logic for all multiplexor and registers that you add. Use symbols like `=rc.reset` to recognize instructions, **show the inputs to these boxes**.

☐ Don't forget the modifications for `movstoi RD, rc.z` and `movstoi RD, rc.nz`.





Problem 2:

(*a*) The code below executes on a dynamically scheduled single-issue (not superscalar) machine using Method 1, register names are reorder buffer entry numbers. The multiply unit is six stages and is fully pipelined (latency 5, initiation interval 1). The add unit has a latency of 3 and an initiation interval of 2. The CDB can handle an unlimited number of writebacks per cycle. Floating point exceptions are **not** precise. (15 pts)

☐ Show a pipeline execution diagram.

☐ Show where instructions commit.

☐ Show changes to the reorder buffer, register map and register file at each cycle. Register f0 initially contains zero, f2 initially contains 20.0; f4, 40.0; etc. Use line numbers for reorder buffer entry numbers.

```
L1:multd  f0, f2, f4

L2:addd   f0, f0, f6

L3:addd   f2, f2, f8

L4:addd   f10, f12, f14
```

☐ The pipeline execution diagram above is the same whether or not exceptions are precise. Why?

(*b*) The code below executes on a two-way superscalar statically scheduled DLX implementation with perfect branch and jump target prediction. (10 pts)

☐ Show a pipeline execution diagram for the worst-case execution of the code below for two iterations. (The worst case is achieved by proper choice of the labels, `LOOP` and `THERE`.)

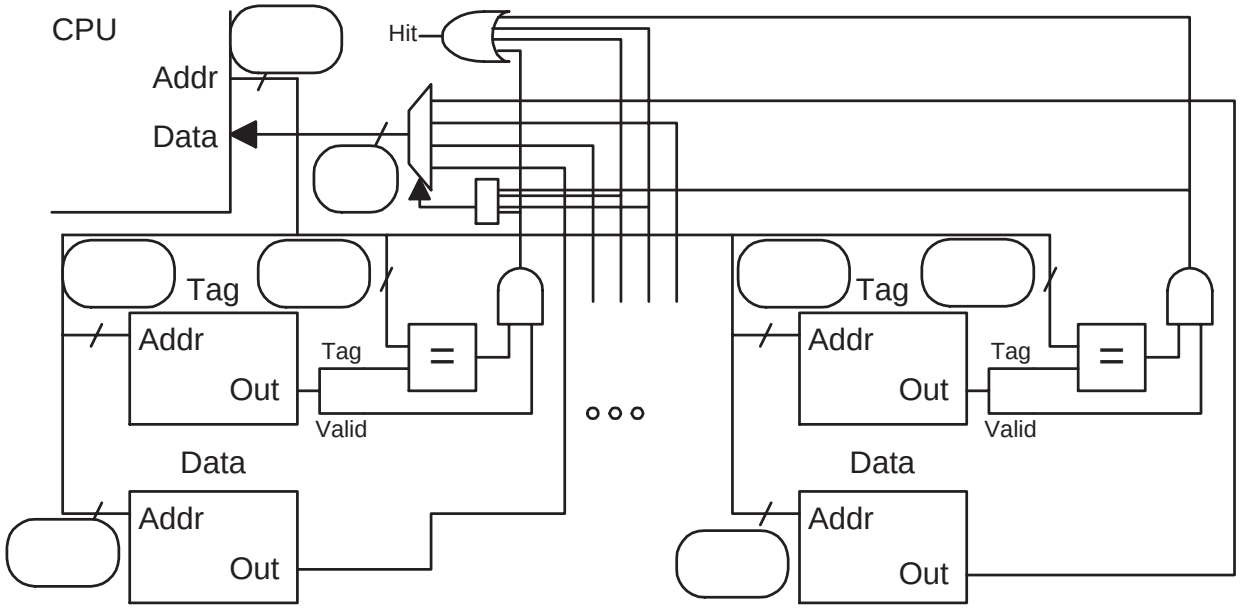
☐ Explain why it's worst case.

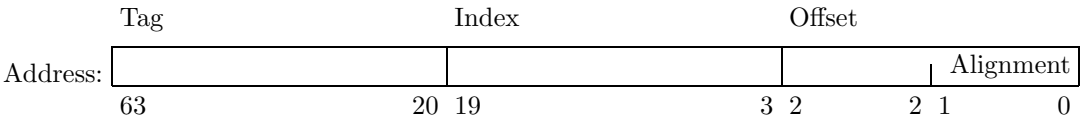
☐ What is the CPI of the for a large number of iterations?

```
 addi r1, r0, #2000
LOOP:
 subi r1, r1, #1
 j THERE
HERE: ! Immediately follows the instruction above.
 bnez r1, LOOP

THERE:
 add r2, r2, r3
 j HERE
```

Problem 3: The cache for a system implementing an ISA with 10-bit characters is described by the following incomplete schematic diagram and address bit categorization.





(*a*) Answer the following, formulæ are fine as long as they consist of literals and grade-time constants. (10 pts)

☐ Fill in the blanks in the diagram.

☐ In the diagram above the processor's data out port is not shown. Given what is shown is the cache probably write back or write through? Argue your answer in terms of what is missing or what is present.

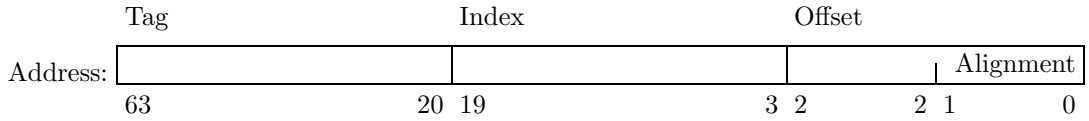
☐ What is the associativity of this cache? (Look carefully.)

☐ What is the capacity of the cache? Specify units!

☐ How much memory does it take to implement the cache? Specify units!

Problem 3, continued: Continue to use the cache from the previous part.

(*b*) When the program below starts execution no data is cached. Consider only memory accesses needed for the array access. The address bits are repeated for convenience. (15 pts)




```
extern char *a;    // & a[ 0 ] = 0x1000000;
int i, j, k, t;
int ISTRIDE = 1024;
for(k=0; k<2; k++) for(i=0; i<256; i++) for(j=0; j<32; j++)
  t += a[ ISTRIDE * i + j ];
```

☐ What is the hit ratio?

☐ What is the smallest line size that would maximize the hit ratio?

☐ What is the smallest *reasonable* line size that would maximize hit ratio?

☐ What is the smallest power-of-two value of ISTRIDE for which a two-way set-associative cache is necessary to avoid conflict misses?

☐ If ISTRIDE were larger than the answer to the previous question but not a power of two, would there be lots of conflict misses? Explain.

Problem 4: Answer each question below.

(*a*) One-level predictors are usually outperformed by two-level predictors. (5 pts)

☐ Write a code fragment containing a branch that would be predicted well by a two-level predictor such as gshare but poorly by a one-level predictor.

(*b*) Virtual memory allows two processes on the same processor to use the same address as though they were on different systems, that is, one process never loads what the other stores. (5 pts)

☐ Why doesn't one process loading from, say, address `0x1000` get data previously stored by another process at address `0x1000`?

(*c*) A processor implementing a 64-bit ISA has the following integer add latencies: 8- and 16-bit integers, 0 cycles; 32-bit integers, 1 cycle; 64-bit integers, 2 cycles; 128-bit integers, 3 cycles. (5 pts)

☐ Why are these latencies not appropriate for a 64-bit ISA? In what important way would the processor suffer?

(*d*) Precise exceptions are optional for floating-point instructions but required for integer instructions. (5 pts)

☐ Why is this so?

☐ Illustrate your answer with an example showing an integer instruction that raises an exception.

☐ Explain what goes wrong if the exception is not precise.

(*e*) What are stops and lookaheads? (5 pts)

☐ How are they used?

☐ What benefit to they have to implementations?

☐ Explain the similarities and differences between stops and lookaheads.