Name Solution

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 Solution!!!

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.*

If the `rs1` field in `movstoi` in the existing instruction is always set to some specified value, say 1, then the existing instruction can be used by indicating that `rs1` specifies which special register to move. A might 1 indicate the special purpose register, a 2 might indicate the new `rc.z` register, and a 3 might indicate the new `rc.nz` register.

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 $\boxed{\text{is Type R}}$ and $\boxed{\text{is Type I}}$ to recognize these instruction types.

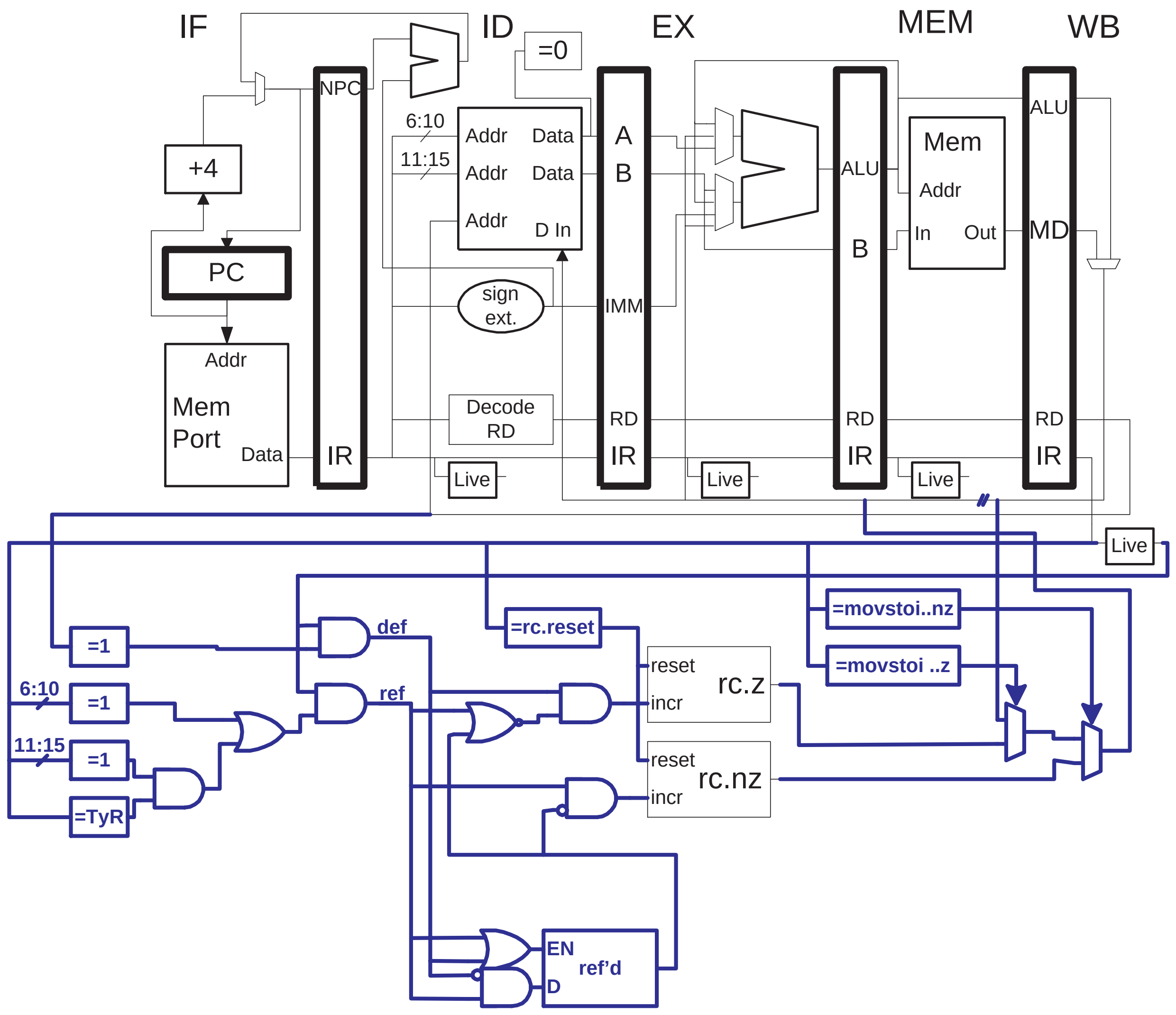
Assume that Decode RD in the diagram can recognize the new instructions

☑ The counts should not include squashed instructions.

☑ Show control logic for all multiplexor and registers that you add. Use symbols like $\boxed{\texttt{=rc.reset}}$ to recognize instructions, **show the inputs to these boxes**.

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

Changes shown in the diagram below in blue. The "ref'd" box is a D flip flop that indicates whether `r1` has been references. Input "EN" is an enable and "D" is the data input.



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
```

```
! Solution
 multd  f0, f2, f4  IF ID M1 M2 M3 M4 M5 M6 WC
 addd   f0, f0, f6     IF ID RS             A1 A1 A2 A2 WC
 addd   f2, f2, f8        IF ID A1 A1 A2 A2 WB             C
 addd   f10, f12, f14        IF ID RS A1 A1 A2 A2 WB          C

 ID Map
 f0     0              L1 L2                             860
 f2     20.0                 L3             100.
 f10    40.0                    L4                260

 Commit File
 f0     0                                   100.        860
 f2     20.0                                           100.
 f10    40.0                                                  260.

 ROB                            L4 L4 L4 L4 L4
                             L3 L3 L3 L3 L3 L3 L4 L4 L4 L4
                          L2 L2 L2 L2 L2 L2 L2 L3 L3 L3 L3 L4
 ROB Head              L1 L1 L1 L1 L1 L1 L1 L1 L2 L2 L2 L2 L3 L4
```

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

Because when a ROB is used for exception recovery there is no need to delay the write back or otherwise change the way long-latency instructions execute.

(*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.

For `LOOP = 0x1004`, the `subi` is the second instruction in an aligned group, so after the first iteration the `addi` and `bnez` will be fetched uselessly. Similar reasoning for `THERE = 0x2004`.

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

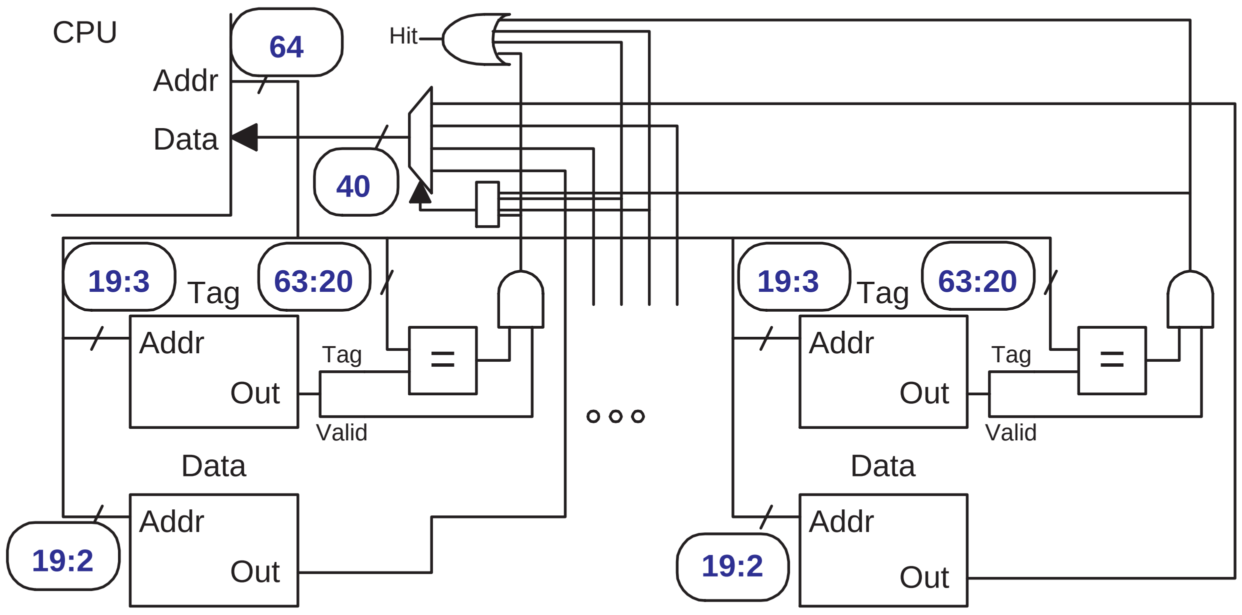
Solution not yet available.

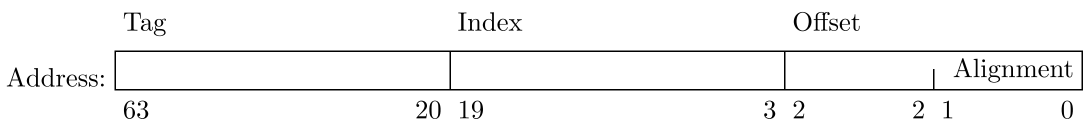
```
 ! Solution not yet available.

 addi r1, r0, #2000
LOOP:
 subi r1, r1, #1
 j THERE
HERE:
 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.

Changes shown in blue.

✓ 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.

Write through because there's no dirty bit. In a write through cache whenever data is written to the cache it is also written to memory. In a write back cache data written to a line in the cache is not written to memory until the line is replaced. A line that has not been modified does not have to be written to memory, the dirty bit is used to distinguish them. It is set to zero when a line is loaded into the cache and it is set to one when the line is written.

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

Four, there are four inputs to the hit or gate.

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

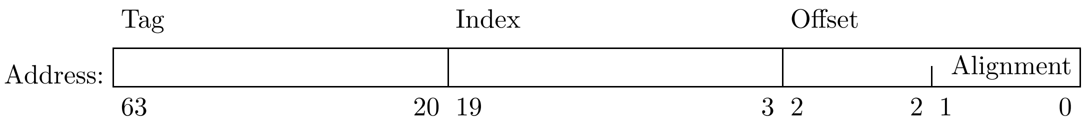
The capacity is $4 \times 10 \times 2^{20}$ bits. The formula above would get full credit, no need to write it in another form such as forty-one million, nine hundred forty-three thousand, forty.

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

The storage used for the tag store is $4 \times 44 + 1) \times 2^{17}$ bits, the total amount of memory is $4 \times 44 + 1) \times 2^{17} + 4 \times 10 \times 2^{20}$ bits.

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?

The hit ratio is $\frac{15}{16}$.

Each line holds eight characters. In the inner loop the array is accessed sequentially. When `k=0` it is being accessed for the first time, and so the hit ratio there will be $\frac{7}{8}$. When `k=1` it is being accessed the second time. Since the entire array fits in the cache the hit ratio is $\frac{1}{2}\left(\frac{7}{8}+1\right)=\frac{15}{16}$.

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

The smallest line size would be 2^{18} characters, which is unreasonably large.

The hit ratio is maximized when the entire array fits on one line. It fits on one line if the tag and index parts of every address are the same. Loop variable `j` "modifies" bits 4:0 of the address. (The address is `a + ISTRIDE * i + j`, where `a` is the address of the first element of the array `a`.) Loop variable `i` modifies bits 17:10. The other bits of the address are the same for every array element. Neither `i` nor `j` modify tag bits, and so the entire array can be cached. The highest bit number modified is bit 17, and so bits 19:18 of the index are the same for every element. If the line size were increased to 2^{18} characters without changing the cache size the index bits would be 19:18, and so the entire array would fit on one line.

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

2^5 characters.

Increasing it to 2^6 would not improve hit ratios; improvements do not occur until it's increased to 2^{11} which is moving in to unreasonable territory.

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

$\texttt{ISTRIDE} = 2^{13}$.

A two-way set-associative cache is necessary when two elements have the same index but different tags. That will happen when exactly one of the bits affected by `i` is part of the tag. For that to happen, $\texttt{ISTRIDE} = 2^{13}$. With that value of ISTRIDE 256 values of `i` yields two different tag values and 128 different indices (ignoring `j`). The elements for `i`=0 have a tag of zero and those for `i`=128 have a tag of 1. For `j`=0 both have an index of 0. As long as it's a power of two `ISTRIDE` shifts `i` to the left.

✓ 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.

There would be few conflict misses because potentially all parts of the address would be affected by `i`, so for 256 different values of `i` there would be 256 distinct index values.

For example, suppose ISTRIDE $= 2^{13} + 2^2$. Now `i` affects bits 20:13 and bits 9:2. There will be a distinct index or offset for each value of `i`.

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.

```
! Solution
 lw r1, 0(r2)  ! r1 hard to predict
 bnez r1, SKIP ! Not predicted well by any predictor.

SKIP:
 beqz r1, SKIP2 ! Since it's correlated with the last branch
                ! a two-level predictor that uses global history will do well.
```

(*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`?

Because the real page numbers in their addresses are different, and so different locations are being accessed.

(*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?

Because 64-bit integer operations are needed to compute memory addresses. Since these are done frequently the processor will suffer.

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

☑ Why is this so?

Because load and store instructions encounter exceptions in normal use (in virtual memory systems) and they must be re-started after exceptions as though nothing happened.

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

```
! Illustration for Answer:
! Cycle:            0  1  2  3  4  5  6  7
lw   r1,0(r2)       IF ID EX *M WB           !  <-  Faulting instruction.
add  r30, r29, r28     IF ID EX ME WB
addi r2, r1, #4           IF ID EX ME WB     !  <-  Last before handler.
and  r27, r26, r25           IF ID EX ME WB  !  <-  Resume here.
```

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

In the example above `lw` raises an exception but `addi` is the last instruction, it executes with a wrong value in `r1`. Execution resumes at the `and` and so the correct value is never put in `r1`.

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

☑ How are they used?

Stops are placed between instructions (in assembler source) to indicate that there is a true dependency between an instruction before the stop and an instruction after the stop. The assembler places this information in an instruction bundle's template.

The lookahead of a bundle, call it x, is the number of following bundles that can be executed before finding a bundle that has a true dependency with bundle x.

☑ What benefit to they have to implementations?

They lower the cost and increase the speed of implementations by eliminating the need for some dependency checking hardware.

☑ Explain the similarities and differences between stops and lookaheads.

They both provide dependency information. Lookaheads only indicate dependencies between bundles, while stops can specify intra-bundle dependencies.