Name Solution

Computer Architecture
EE 4720
Final Examination

13 December 2005, 12:30–14:30 CST

Problem 1		(15 pts)
Problem 2		(20 pts)
Problem 3		(17 pts)
Problem 4		(15 pts)
Problem 5		(33 pts)
Exam Total		(100 pts)

Alias Out-of-order graduation?

Good Luck!

Problem 1: The implementation of a version of MIPS with loop count instructions is shown below, these are the same instructions used in Homework 5. Instruction `mtlc rt` moves the contents of the `rt` register into a special loop count register, `lc`, and instruction `mtlci immed` moves a 16-bit immediate, `immed`, into `lc`. The loop-counted branch `bclz` is taken unless the `lc` register is zero, it also decrements `lc`. (In the diagram the upper input to the `lc` register is the data input and the lower input, `we`, is a write enable.)

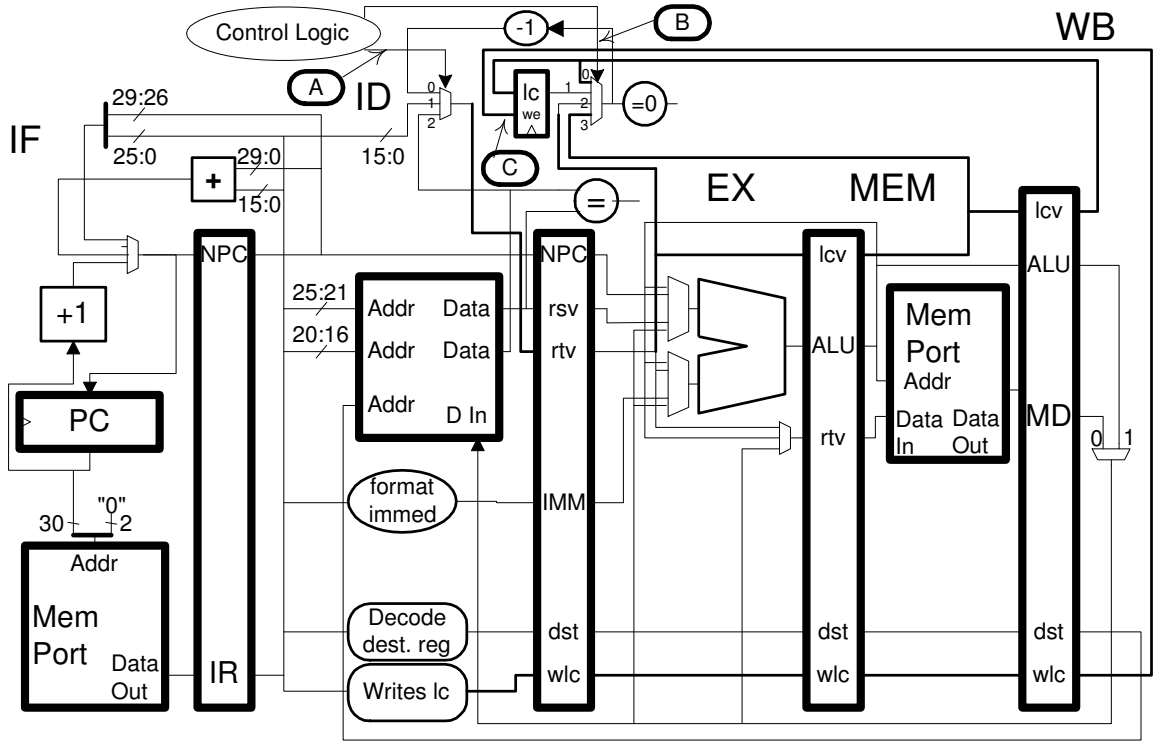
Three wires in the implementation are labeled (A , B , and C). Show the values present on those wires for each cycle of the illustrated execution of the program in the space provided. Leave a value blank if the value has no effect, assume that instructions before and after the illustrated code are nops. Note that two iterations of the loop are shown. (15 pts)

```
 # Cycle            0    1    2    3    4    5    6    7    8
 mtlci 100          IF   ID   EX   ME   WB
LOOP:
 bclz LOOP               IF   ID   EX   ME   WB
 add r2, r2, r3               IF   ID   EX   ME   WB
# (2nd iteration)
 bclz LOOP                         IF   ID   EX   ME   WB
 add r2, r2, r3                    IF   ID   EX   ME   WB
 # Cycle            0    1    2    3    4    5    6    7    8

 A:                      1    0    2    0    2    0*   2*   0*   <- SOLUTION
 B:                           2         3         3*        3*   <- SOLUTION
 C:                 0    0    0    0    1    1    0    1    0    <- SOLUTION

 # Cycle            0    1    2    3    4    5    6    7    8
```

See next page for discussion of solution.



Discussion of solution on previous page.

The loop count branch examines the `lc` register (or a bypassed value) when it is in the ID stage. (If it examined it in a later stage the first branch would have to stall.) Control signal B selects whether the `lc` register itself is used, 1, or a value in EX, 2, in MEM, 3, or in WB, 0. Register `lc` is updated by `bclz`, `mtlc`, and `mtlci`, but to maintain precise exceptions the update is done in the WB stage. The new value for the `lc` register passes through the existing `ID/EX.rtv` latch, then goes through two new latches, `EX/MEM.lcv` and `MEM/WB.lcv`. Wire A controls an ID-stage mux that selects which value to send to the `ID/EX.rtv` latch. A 2 on A selects the rt value, this would be used for `mtlc` and also for existing type R and store instructions. A 1 selects the immediate, for `mtlci`, and a 0 selects the decremented `lc` value, for `bclz`. Signal C is the write-enable for the `lc` register.

For the `mtlci` instruction A is set to 1 in cycle 1 to select the immediate. The value for B in cycle 1 is left blank because the value of `lc` is not being used. When `mtlci` reaches writeback, in cycle 4, C is set to 1, writing the immediate to lc.

Signal C is set to 1 when the `bclz` instructions reach writeback, cycles 5 and 7 and should be 0 at other times to avoid overwriting a good value. That is, in a correct solution a blank could not appear anywhere for C.

When `bclz` is in ID signal A is 0 to select the decremented value for writeback, that's in cycles 2 and 4. In cycle 2 B is set to 2 to bypass the `lc` value from EX (written by `mtlci`); in cycle 4 B is set to 3 to bypass the `lc` value from MEM (written by the previous `bclz`). If the loop proceeds for another two iterations B would be set to 3 in cycles 6 and 8. All other values for B should be blank because the output of the mux will be ignored in those cycles.

The `add` instruction is in ID in cycles 3 and 5 at which time A is set to 2 to pass the rt register value. If the loop continues A is 2 in cycle 7 and 0 in cycles 6 and 8.

Problem 2: Complete the pipeline execution diagrams for the different MIPS implementations below. Pay attention to the type of implementations in each part, they're not all the same. Don't forget to **check for dependencies.**

☑ (5 pts) Scalar (not superscalar), statically scheduled, as illustrated in the previous problem.

```
# SOLUTION
#
# Cycle           0  1  2  3  4  5  6  7
 add r1, r2, r3   IF ID EX ME WB
 lw r4, 6(r1)        IF ID EX ME WB
 xor r7, r4, r8         IF ID -> EX ME WB
# Cycle           0  1  2  3  4  5  6  7
```

In all of the parts below the floating-point multiply unit has four stages and is fully pipelined, the stage labels are *M1-M4*. *The floating-point add unit is two stages and fully pipelined, the stage labels are* *A1 and A2*.

☑ (5 pts) Scalar, statically scheduled, full set of bypass paths.

```
# SOLUTION
#
# Cycle                0  1  2  3  4  5  6  7  8  9  10 11 12 13
 mul.d f2, f4, f6      IF ID M1 M2 M3 M4 WF
 add.d f8, f2, f10        IF ID -------> A1 A2 WF
 mul.d f2, f4, f14           IF -------> ID M1 M2 M3 M4 WF
 sub.d f12, f2, f10                      IF ID -------> A1 A2 WF
# Cycle                0  1  2  3  4  5  6  7  8  9  10 11 12 13
```

Common mistake:

Many solutions showed a MEM stage. The MEM stage is only used in the integer pipeline.

Problem 2, continued:

☑ (5 pts) Two-way superscalar, statically scheduled, full set of bypass paths. No alignment restriction on instruction fetches.

```
# SOLUTION
#
# Cycle               0  1  2  3  4  5  6  7  8  9  10 11 12 13
mul.d f2, f4, f6      IF ID M1 M2 M3 M4 WF
add.d f8, f2, f10     IF ID ----------> A1 A2 WF
mul.d f2, f4, f14        IF ----------> ID M1 M2 M3 M4 WF
sub.d f12, f2, f10       IF ----------> ID ----------> A1 A2 WF
# Cycle               0  1  2  3  4  5  6  7  8  9  10 11 12 13
```

Common Mistake:

Some did not show a stall for the second multiply. There is room in ID for the second multiply in cycles 2-5, but it would require more complex control logic to accommodate it (since the instructions in ID would be out of order) and so the default behavior in this class is a stall.

☑ (5 pts) Two-way superscalar, dynamically scheduled, with a full set of bypass paths. No alignment restriction on instruction fetches.

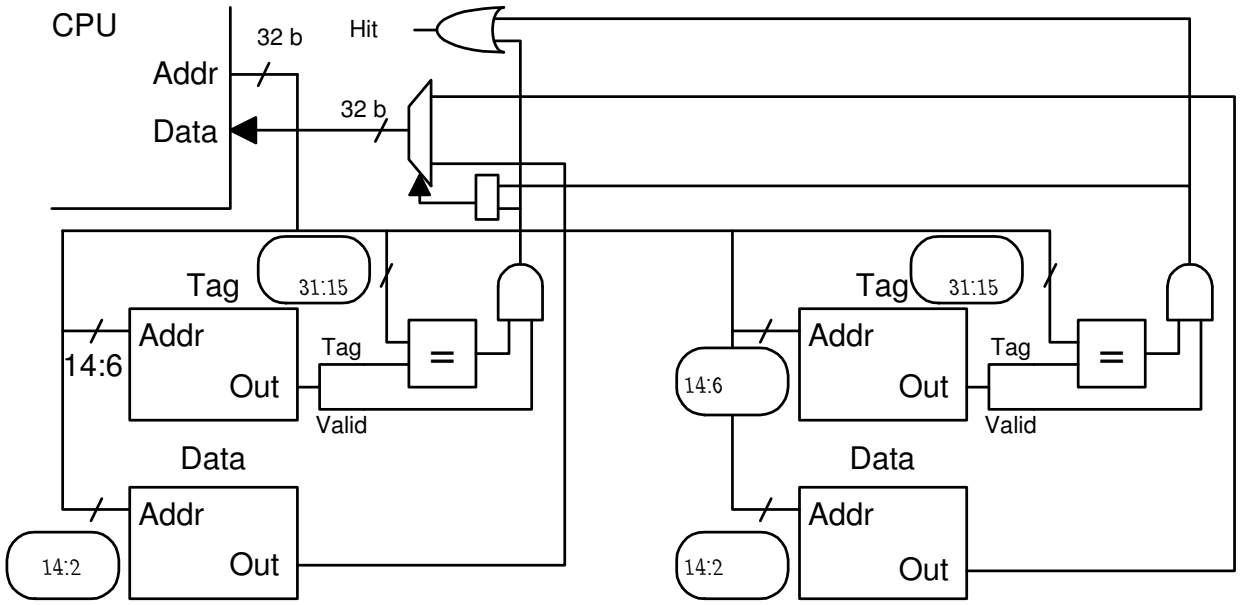
```
# SOLUTION
#
# Cycle               0  1  2  3  4  5  6  7  8  9  10 11 12 13
mul.d f2, f4, f6      IF ID Q  RR M1 M2 M3 M4 WB C
add.d f8, f2, f10     IF ID Q              RR A1 A2 WB C
mul.d f2, f4, f14        IF ID Q  RR M1 M2 M3 M4 WB    C
sub.d f12, f2, f10       IF ID Q              RR A1 A2 WB  C
# Cycle               0  1  2  3  4  5  6  7  8  9  10 11 12 13
```

Not-too-common-but-more-common-that-it-should-have-been-mistake: Some solutions showed the second multiply waiting in one way or another for the add. There is no reason for the wait, this is a dynamically scheduled processor.

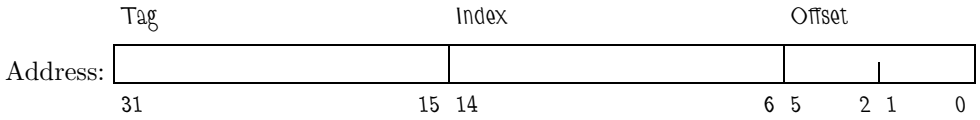
Problem 3: The diagram below is for a cache on a system with the usual 8-bit characters.

(*a*) Answer the following, formulæ are fine as long as they consist of grade-time constants. (7 pts)

☑ Fill in the blanks in the diagram.



☑ Show the address bit categorization. Label the sections appropriately. (Alignment, Index, Offset, Tag.)



☑ Cache Capacity (Indicate Unit!!):
Capacity is 2×2^{15} characters (64 kiB).

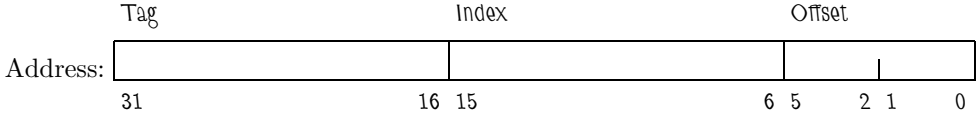
☑ Memory Needed to Implement (Indicate Unit!!):

It's the cache capacity plus $2 \times 2^{15-6}\,(32 - 15 + 1)$ bits.

☑ Line Size (Indicate Unit!!):

Line size is $2^6 = 64$ characters.

☑ Show the bit categorization for a direct mapped cache with the same capacity and line size.



Problem 3, continued: For the problems on this page use the cache from the previous page.

(*b*) The code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☑ (5 pts) What is the hit ratio running the code below?

```
double sum = 0.0, *a = 0x2000000;  // sizeof(double) = 8 characters
int i, j;

for(j=0; j<3; j++)
  for(i=0; i<32; i++)
    sum += a[ i ];
```

The code above sequentially accesses elements of size 8 characters and so there are $64/8 = 8$ elements per line. On the first `j` iteration the first access to an element on a line will miss and the others will hit, for a hit ratio of $\frac{7}{8}$. Only four lines will be used which is much smaller than the cache capacity and so on the second and third `j` iterations all access will hit. The overall hit ratio is $\boxed{\frac{7+8+8}{8+8+8} = \frac{23}{24} = 0.95833333}$.

(*c*) The slightly different code below runs on the same cache as the first part of this problem. Initially the cache is empty; consider only accesses to the array.

☑ (5 pts) Set `ILIMIT` to the smallest value that will fill the cache.

```
double sum = 0.0, *a = 0x2000000;  // sizeof(double) = 8 characters
int i;

int ILIMIT = 1 << 13; // 1 << 13 == 8192     <- SOLUTION


for(i=0; i<ILIMIT; i++)
    sum += a[ i ];
```

The cache capacity was found to be 2^{16} characters. The code above starts on a nice round address (many low bits are zero) and accesses data sequentially and therefore does not waste any space on a line. Each array element is 2^3 characters and so the cache will be exactly filled if $\frac{2^{16}}{2^3} = 2^{13} = 8192$ elements are accessed. (For those rusty on their C operators, `<<` is the left-shift operator.)

Problem 4: Answer each question below.

(*a*) An n-way superscalar processor need not have n copies of everything that an ordinary scalar implementation has.

☑ (5 pts) Which of the following is it most important that an n-way superscalar processor **does** have n of (explain):

- Mem-stage memory ports.
- Write ports to the register file.
- Floating-point adders.

Most important: Write ports to the register file.

Suppose a 2-way statically scheduled superscalar processor had one memory port. Then it would stall whenever a fetch group had two memory instructions. Similarly, such a system with two write ports to the register file would stall whenever a fetch group had two instructions that write the register file. Since more instructions write the register file (in fact, most of them) than use the MEM-stage memory ports it's better to have two register write ports. The same argument can be made for the floating-point adder.

The answer would be the same for a dynamically scheduled system but the argument would be slightly different.

(*b*) In the dynamically scheduled MIPS implementation covered in class:(5 pts)

☑ When is a physical register allocated (removed from the free list and assigned to an instruction)?

When an instruction is in ID.

☑ When is the physical register put back in the free list. Show an example that includes register numbers.

When the next instruction that writes the same architected register commits. In the example below a physical register (say `p123`) is assigned to the `add` when it is in ID, in cycle 1. The `xor` is the next instruction that writes the same architected register as the `add`, `r1`. When `xor` commits, in cycle 8, register `p123` is put back in the free list.

```
# Cycle           0  1  2  3  4  5  6  7  8
 add r1, r2, r3   IF ID Q  RR EX WB C
 sub r9, r1, r10     IF ID Q  RR EX WB C
 xor r1, r4, r6         IF ID Q  RR EX WB C
# Cycle           0  1  2  3  4  5  6  7  8
```

(*c*) Describe the contents of an Itanium (IA-64) bundle or a bundle in a typical VLIW ISA.

☑ (5 pts) Bundle Contents

In the Itanium ISA a bundle contains three 41-bit instructions and 5 template bits. The template bits specify whether the instructions are dependent and also the functional units that they will use.

For a typical ISA a bundle contains several instructions (three is a popular number) along with some dependency and possible other information about the instructions.

Problem 5: Answer each question below.

(*a*) When a company wants to benchmark a new computer using SPECcpu it gets the benchmark suite from SPEC, compiles the code, runs the benchmarks, and proudly publishes the results. (The last step is optional.)

Below are two variations on this procedure, for each explain how the results might differ **from those obtained using the usual SPEC procedure**, and explain how useful the results would be to the typical user of SPECcpu results. (7 pts)

The key difference between the scenarios below and the standard testing procedure is in who chooses the compiler and in who runs the compiler; normally the company does both. The choice of compiler is important because the full potential of the processor may only be realizable with specially tailored compiler optimizations and code generation. It is in the company's interest to develop such a compiler and its development should go hand-in-hand with developing the processor implementation used in the new computer. SPEC's interest is in having useful benchmark results so if they did choose a compiler they would try to pick one suitable for the computer, but they lack the company's expertise and they certainly would not take the trouble to write one. (In reality they would ask the company to provide one, but in this problem that doesn't happen.) So, if SPEC were to choose the compiler it may not choose one best suited to the processor and so the results may not be as good.

Users who compile their own code would probably use the company's compiler to get the best results and so procedure (1) would not be very realistic.

In procedure (2) the company's compiler is used but SPEC is doing the compilation. The company may have had well-motivated experts doing the compilation who would have gotten better results than SPEC's not-as-expert and not-as-motivated people. The base scores might not differ because only basic optimizations should be used and so SPEC and the company would compile identically. Whether the peak ("result") results would be better than procedure (1) depends on how important the compiler is. (Either answer would be correct if properly justified.) The results would certainly be more useful than (1) for users since if more closely reflects how they would develop code.

(1) SPEC gives a compiler (in addition to the usual material) to the company and allows the company to compile and run the suite following the usual SPECcpu rules (but using SPEC's compiler).

☑ Compared to actual SPEC rules and to (2), how might the results differ? Explain.

Not as high as actual SPEC or (2) because the company can produce a much better compiler. (See discussion above.)

☑ Compared to actual SPEC rules and to (2), how useful would results be? Explain.

Not as useful as actual SPEC or (2) because real users would use company's compiler. (See discussion above.)

(2) The company gives the new computer and a compiler to SPEC and SPEC compiles and runs the benchmark suite using the usual SPECcpu rules.

☑ Compared to actual SPEC rules and to (1), how might the results differ? Explain.

The results would be better than (1) because a better compiler is used but not as good as actual SPEC because tuning not as good as the company's.(See discussion above.)

☑ Compared to actual SPEC rules and to (1), how useful would results be? Explain.

Results would be more useful than both (1) and actual SPEC because they were obtained by skilled people but not super-motivated experts, so one might expect to get similar performance with reasonable skill and expertise.(See discussion above.)

(*b*) There is much less advantage in unrolling one of the loops below when the code is to run on a two-way statically scheduled superscalar MIPS implementation. *Note: The original problem did not mention the code was to run on a superscalar implementation.*(6 pts)

☑ Unroll the loop below or explain why it shouldn't be unrolled.

```
 addi r9, r0, 1024
LOOP_A:
 lw r1, 0(r2)
 lw r1, 0(r1)
 add r3, r1, r3
 addi r2, r2, 4
 bneq r9, r0 LOOP_A
 addi r9, r9, -1
```

Unrolling loops provides more flexibility to schedule away stalls. In the code above the stalls are between the first two loads and the load and the add. These can be scheduled away on a scalar implementation but not on a 2-way superscalar.

Unrolling only works if the only dependencies between iterations are carried by things like steadily increasing loop indices, such as those carried by `r2` and `r9` in the loop above. To unroll the loop twice two iterations of the original loop are made one iteration of the new loop; that's shown below. On a scalar system all stalls are eliminated, the 2-way superscalar system would still have stalls so it would have to be unrolled four times.

```
# SOLUTION
 addi r9, r0, 1024
LOOP_AU:
 lw r1, 0(r2)
 lw r10, 4(r2)
 lw r1, 0(r1)
 lw r10, 0(r10)
 add r3, r1, r3
 add r13, r10, r13
 addi r2, r2, 8
 bneq r9, r0 LOOP_AU
 addi r9, r9, -2
 add r3, r3, r13
```

☑ Unroll the loop below or explain why it shouldn't be unrolled.

```
 addi r9, r0, 1024
LOOP_B:
 lw r1, 0(r2)
 add r3, r3, r1
 lw r2, 4(r2)
 bneq r9, r0  LOOP_B
 addi r9, r9, -1
```

There is not much benefit in unrolling the loop above because `r2` used in the beginning of an iteration is produced by a `lw` from the previous iteration so there is no way two iterations could overlap.

(*c*) Which are smaller, RISC programs or CISC programs. Provide some instruction examples to illustrate your answer. (5 pts)

☑ Example and explanation.

CISC programs are smaller because their instructions, being variable size, do not waste space. Also, instructions can do more such as accessing operands from memory, doing arithmetic, and storing the result back in memory. In the example below one CISC instruction does the work of four RISC instructions. Three of the RISC instructions waste 16-bits on an unused immediate. In additional the RISC code uses four opcodes and uses space for 9 register numbers, while the CISC code uses one opcode and space for three registers.

BTW, the size advantage of CISC is outweighed by the disadvantage of more complex implementations so don't get the wrong idea.

```
# SOLUTION EXAMPLE

# CISC
 add (r1), (r2), (r3)

# RISC
 lw r4, 0(r2)   # Load instruction wastes 16 bits on an immediate thats zero.
 lw r5, 0(r3)   # Ditto
 add r6, r4, r5
 sw r6, 0(r1)   # Also wastes 16 bits.
```

(*d*) The cost of implementing a rarely used instruction is deemed too high and so is omitted from an implementation. How can an operating system enable the implementation to run code that uses the rare instruction without modifying the code or examining it in advance. (5 pts)

The rarely used instruction will raise an illegal instruction exception (since it's not implemented). The operating system will have the handler check the opcode and if it's the rarely used instruction the handler will do exactly what the instruction would (reading the same registers, computing the result, and writing the result in the correct register) using implemented instructions.

(*e*) BCD data types were once popular ISA features. (5 pts)

☑ Were BCD data types absolutely necessary?

No, could use integer data types but that would be inconvenient. When it comes to computers—and many other areas—very few things are absolutely necessary.

☑ What were the reasons for including BCD data types in an ISA?

It eliminated the need to convert to decimal, as when preparing human-readable output. It also allowed for the exact representation of fixed point decimal numbers and eliminated rounding errors in many calculations.

☑ Why might a BCD data type be considered old fashioned and so not worthy of adding to an ISA now?

Computers are much faster and so decimal conversion is very fast. With floating-point standards, in particular IEEE 754, rounding errors are predictable and are minimized in calculations using fixed point decimal numbers.

(*f*) MIPS-I is big endian but more recent MIPS versions can run in either big-endian or little-endian modes.

☑ (5 pts) Complete (possibly modifying) the MIPS program below so that it writes `v0` with a 0 if it is running on a system using big endian byte order or a 1 if running on a system using little-endian byte order.

```
# At finish: $v0: 0, if big endian;  1, if little endian.
lui $t1, 0x1122
ori $t1, $t1, 0x3344
sw $t1, 0($t2)
```

```
# SOLUTION
# At finish: $v0: 0, if big endian;  1, if little endian.
ori $t1, $0, 0x1
sw $t1, 0($t2)
lb $v0, 0($t2)
```

Note that the operands for `ori` were changed and `lui` was eliminated. Byte order **does not** effect non-memory instructions, including `lui` and `ori`. The `ori` sets `t1` to `1` which is written to memory. On a little-endian system the least-significant 8 bits of `t1`, `0x01`, is written to address `t1`, the next 8 bits, `0x00`, are written to `t1+1`, etc. On a big-endian system the most-significant 8 bits, `0x00`, is written to `t1`. If the `lb` reads a `0x01` it must be a little endian system.

Grading Note: Many solutions were correct but most were more complicated than they needed to be, some using branches.