Question 1 or 2:Parts of a Computer (3 points)

Jonathan and Kurt

a)

```
control
datapath
memory
input
output

Minus one point for every answer that wasn't there, up to a maximum of
two. We

grudgingly accepted 'CPU' for control.
```

b)

We accepted just about any answer that was remotely reasonable. One point was given if at least 3 of the 5 made sense.

Question 2 or 5:Pointers (5 points)

Jonathan and Kurt

All the parts were graded on an all or nothing basis.

a)

```
'la $s0, x'
'li $s0, 1000'
```

We accepted anything that loaded 1000 into $s0.

b)

```
'la $s1, y'
'li $s1, 1004'
```

We accepted anything that loaded 1004 into $s0.

c)

```
'lbu $t0, 0($s0)
sb $t0, 0($s1)
addiu $s0, $s0, 1'
```

Question 1 or 2:

We also accepted 'lb' for 'lbu'. However, we did not accept 'addi' for 'addiu'.

d)

```
'mov $s1, $s0'
'add $s1, $s0, $0'
```

We accepted anything that copied $s0 into $s1.

e)

1001

Question 3 or 1:Pliable Data (5 points)

Kurt, Jonathan

The first three parts were graded on a all or nothing basis.

a)

0010 0000 1100 1000 10001 0000 0000 0000

b)

2^12 + 2^15 + 2^19 + 2^22 + 2^23 + 2^29

c)

[1 + 2^(-1) + 2^(-4) + 2^(-8) + 2^(-11)] x 2^(-62)

d)

```
'addi $8 $6 -28672'
'addi $t0 $a2 -28672'
```

One point for addi and the registers. One point for the immediate. We accepted any signed sum of powers of two that added to -28672. The most succinct was 2^12 - 2^15.

Question 4 or 6:Can a C integer float? (2 points)

Question 1 or 2:

Steve V and Sumeet

a)

No. There are only 23 bits of precision in single precision floats, but the int type uses 32 bits to represent integers. (1pt)

b)

Yes. There are 52 bits of precision in double precision floats which is more than enough needed to represent 32 bit integers. (1pt)

Many students felt all integers could be recovered from single precision float because floats have a larger range of values than ints. This is true, but single precision floats sacrifice precision for range.

Many students also felt that integers could not be recovered from single precision floats because floating point numbers are an approximation and don't have the precision necessary to represent integers. We accepted an answer like this, but only if you mentioned something about the precision (and not the range!). the above is only true for single precision floats. Some people claimed that integers couldn't be recovered from doubles for the same reason. This is false! Doubles have more than enough precision to represent 32 bit integers. To get credit for part b.), you needed to mention why doubles would give you more precision than singles (i.e. just saying that doubles are more precise is insufficient).

Question 5 or 4:Networks (10 points)

David and Steve Tu

We will sometimes abbreviate "bytes" as **B**. We will sometimes abbrevate "seconds" as **sec** or **s**.

I.

For (a) and (b), the important thing to understand is that there are three components to the time it takes to send a message from A to B. The first is the constant per-call overhead when you call send(), which is 1x10^-3 seconds per call. The second is the per-byte overhead when you call send(), which is 1x10^-5 seconds per byte. The third is the amount of time it takes the data to travel over the wire from A to B (calculated from the network bandwidth), which is 5x10^5 bytes/second or 2x10^-6 seconds/byte.

(a)

The "first way" of sending the data involves calling send() 500 times, each time sending 4x10^3 bytes. Adding up the three components, we have

500 calls * [(1x10^-3 s/call) + (1x10^-5 s/byte)(4x10^3 bytes/call) + (4x10^3 bytes/call)/(5x10^5 bytes/sec)]
= 24.5 sec.

Note that the wording of the problem did not technically require you to count the amount of time it takes the data to travel over the wire from A to B, so we also gave full credit for

500 calls * [(1x10^-3 s/call) + (1x10^-5 s/byte)(4x10^3 bytes/call)] = **20.5 sec.**

(b)

The "second way" of sending the data involves calling send() 1 time, sending 2x10^6 bytes all at once. Adding up the three components, we have

1 call * [(1x10^-3 s/call) + (1x10^-5 s/byte)(2x10^6 bytes/call) + (2x10^6 bytes/call)/(5x10^5 bytes/sec)] = **24.001 sec.**

Note that the wording of the problem did not technically require you to count the amount of time it takes the data to travel over the wire from A to B, so we also gave full credit for

1 call * [(1x10^-3 s/call) + (1x10^-5 s/byte)(2x10^6 bytes/call)] = **20.001 sec.**

II.

For (c) and (d), we add two new components to the amount of time needed to send a message from A to B. The first new component is the constant per-message time it takes B to generate an acknowledgment message, which is 1 x 10^-3 sec, and the second new component is the amount of time it takes the acknowledgment message to travel over the wire from B to A (calculated from the network bandwidth), which is 5x10^5 bytes/second or 2x10^-6 seconds/byte. There is one acknowledgment message sent by B for each send() call that A does.

(c)

The extra time added by the generation and transmission of the acknowledgment is

500 acknowledgment messages * [1 x 10^-3 s/message + (1 x 10^4 bytes/acknowledgement message)/(5x10^5 bytes/s)] = 10.5 sec.

Therefore the total time is whatever you got in part (a) plus 10.5 sec, so we accepted 24.5+10.5 = **35.0 sec** or 20.5+10.5 = **31.0 sec**.

(d)

The extra time added by the generation and transmission of the acknowledgment is

1 acknowledgment message * [1 x 10^-3 s/message + (1 x 10^4 bytes/acknowledgement message)/(5x10^5 bytes/s)] = 0.021 sec .

Therefore the total time is whatever you got in part (b) plus 0.021 sec, so we accepted 24.001 + 0.021 = **24.022 sec** or 20.001 + 0.021 = **20.022 sec**.

Question 1 or 2:

Note that one popular mistake was to also add the two components of send() call overhead when B generates an acknowledgment message. The question explicitly stated that B generates the acknowledgment messages directly on its network interface, without involving the CPU or memory of B. Therefore this send() call overhead is not part of the time B takes to generate its acknowledgment messages (though certainly it is still part of A sending the true message). We only took off one point in each of parts (c) and (d) if you did this and otherwise got the problem correct. If your work was sufficiently clear, we did not take off points for arithmetic errors. Sufficiently clear means we saw correct formulas and plugged-in numbers. Insufficiently clear was writing pseudo-random numbers on the page, of unknown origin, without symbols like '+' or '('.

Question 6 or 3:Starting a Program (2 points)

Grading

Jonathan and Kurt

2: EACDB

1: EACDB in wrong order

0: inability to correctly generate first five letters of alphabet, e.g. JQFMD, 10000, strange doodlings and/or bite marks

Reasoning

You write your C code first (E). 'Nuff said about that. The compiler, in this case, produces MAL. Compilers don't have to produce MAL, but it's pretty obvious that the compiler for this problem does use MAL, not TAL. Anyway, the assembler has to convert this MAL to TAL (A) before it can output the binary representations to the .o files (C). Assemblers don't necessarily convert the MAL text to TAL text, but they must translate pseudocode, at least in some abstract way. After the .o files are produced, the linker patches internal (e.g. jal's within the file) and external (e.g. jal's to other files) references (D). The result is the executable file (B).

Anticipated complaints

C1) Technically, the .o files can be "produced," that is, created or opened (but not completely written) _before_ the MAL is converted to TAL!

A1) Please. If we ask you to "produce" proj5 and all you do is open the file in emacs, we're going to give you a low grade. You got the wrong answer, so you got a 1.

C2) What if M'Piero recompiled multiple times? The order could be totally jacked, so any answer is correct.

A2) Look, anyone who has an ALU for feet is going to get code to compile on the first try.

C3) This question was so vague!

Question 1 or 2:

A3) What is wrong with you?

* This problem reminded you that addiu sign-extends the immediate, a hint for the pliable data question.

Question 7:MIPS (9 points)

Sumeet and Steve V.

a)

$t1= no answer / always changing / where ever you stop it **1 pt**.

Why not?Infinite loop: $t3 never modified, $s0 never modified. **2 pt.**

b)

$t1=Must have correct reasoning to get points here.

0x50000 **2 pt.**

0x40000,0x60000 -- off by one **1 pt.**

0x10000 -- With incorrect field being modified **2 pt.**

0x80000 -- Incorrect field and Not seeing $at != 0 **1 pt.**

Why not? Self modifying code **2 pt**.

Branch instruction is modified **1 pt.**

Correct field of branch instruction **1 pt.**

Common Mistakes

It's absolutely amazing how many people thought that the instruction sw $2,0($s0) modified the value in $s0. Sw takes the value in its first argument, and stores it in the memory address calculator by adding the constant to the value in the second register. The constant is not added into the register. The first register is not added to the second register. I.E. $s0 has the same value all the way through the program!

Question 8:Pointers in C and MIPS (14 points)

Solution, Lan 'n Dan

Underlined lines are the lines you had to fill it. We also gave you the option of writing your own solution from scratch, though I don't recall seeing anyone who chose this option.

```
findData:
addi $sp, $sp, -28
sw $a0, 16($sp)
sw $a1, 20($sp)
sw $ra, 24($sp)
jal hashFunction
lw $a0, 16($sp)
lw $a1, 20($sp)
lw $ra, 24($sp)
addi $t0, $0, TABLE_SIZE # Pretend TABLE_SIZE is a constant
divu $v0, $t0
mfhi $t0  # $t0 = hashFunction(key) % TABLE_SIZE
add $t0, $t0, $t0 # $t0 = 2 x location
add $t0, $t0, $t0 # $t0 = 4 x location
add $t0, $t0, $a1 # $t0 = &hashTable[location]
lw $t0, 0($t0) # $t0 = hashTable[location]
loop:
beq $t0, $0, not_found  # while(lookAtMe != NULL)
lw $t1, 0($t0)  # $t1 = lookAtMe->key
bne $t1, $a0, next_bucket  # if(lookAtMe->key == key). . .
 # The above comment is somewhat
# misleading
 # if(lookAtMe->key != key)
 # might be more accurate
lw $v0, 4($t0)  # $v0 = lookAtMe->value
j fin
next_bucket:
lw $t0, 8($t0) # lookAtMe = lookAtMe->next
j loop
not_found:
addi $v0, $0, ERROR_NOT_FOUND # Pretend ERROR_NOT_FOUND is a
 # constant
fin:
addi $sp, $sp, 28
jr $ra
```

Grading

The problem was out of 14 points. If your solution was correct, you got all 14 points. Otherwise, we assigned 2 points for each correct line you filled in. If you had a line basically correct, but got a single register wrong or made a similar typo-esque mistake, we docked you 1 point so you'd get 1/2 points for the line.

Question 1 or 2:

If you had an unorthodox solution (i.e. one that didn't follow our rubric above), then you got 14 points if it was correct. If it wasn't quite correct, then we did our best to assign you as many points as was possible based on what it seemed you were doing. In all likelihood, we were too generous when doing this, so be wary if you ask for a regrade as you may end up with lower score than you originally had.

Common Errors and Grading Oddities

Multiplying the array index by four

We felt that the important thing here was that you understood that hashTable was an array of pointers, and that, for the purposes of this class, pointers are 32 bits (thankfully, nobody tried to use 64-bit pointers). Consequently, you needed to multiply the array index by four. There are a number of ways to accomplish this. We accepted all of them.

The solution above accomplishes the multiplication using two add statements. In this case, each add statement netted you 1 point (for a total of 2). Some people used a single sll statement, which also got 2 points. Some people used mul, mult, and multi (which doesn't even exist). We begrudgingly gave most of these solutions the full 2 points as well, since they addressed the concept, though, more often than not, they munged the assembly

Understanding the meaning of **hashTable

A lot of folks didn't understand what **hashTable meant when we said findData(int key, HashBucket **hashTable). Since we gave you the declaration for the hashtable, namely HashBucket *hashTable[TABLE_SIZE], you should have seen that the hash table was an array of pointers to HashBuckets.

Some folks made the mistake of thinking that hashTable was an array of HashBucket structs (i.e. not an array of pointers). If you did this, we deducted 3 points from your solution because we felt this was a pretty significant misunderstanding.

Others dereferenced hashTable too many times (i.e. performing a second lw prior to the loop label). For this, we deducted 2 points.

Misordering Statements

Most statement misorderings resulted in a 1 point deduction. If you swapped the ordering of the beq and lw statements in the loop, we deducted 2 points because this suggested that you didn't understand the difference between lookAtMe the pointer, and lookAtMe->key, the field in the struct lookAtMe points to.

Bogus Statements

If you had a statement that made an otherwise correct solution incorrect, we deducted 1 point. No deduction was made if you wrote a statement that had no effect.