CS61C Fall 2013 Midterm Instructions

This exam is closed book, closed notes, open two-sided crib sheet. Please put away all phones and calculators -- you won't need them. The exam is worth **50** points and represents 10% of your course grade. It contains **50** multiple choice questions on **14** numbered pages, including the cover page. One and only one answer is correct for each question. There is no penalty for incorrect answers. The MIPS Green Card is appended at the end. Be sure to write your name, course login, and lab section on the separately distributed answer form.

Don't cheat by looking at other students' answer sheets. Penalties will be severe if you are caught. Furthermore, there is absolutely no guarantee that your neighbor knows what they are doing!

Section I: *Cloud Computing and Warehouse Scale Computers*

1. Consider what happens to Power Usage Effectiveness (PUE) when the total energy consumed by a datacenter stays the same but its IT equipment becomes more energy efficient by cutting its energy consumed in half? Which answer best describes the effect on PUE?
   (a) PUE is halved
   (b) PUE doubles
   (c) PUE stays the same
   (d) PUE approaches 1
   (e) PUE approaches 0

Since PUE is (IT Power)/(Total DC Power), having the numerator has the effect of doubling the whole expression.

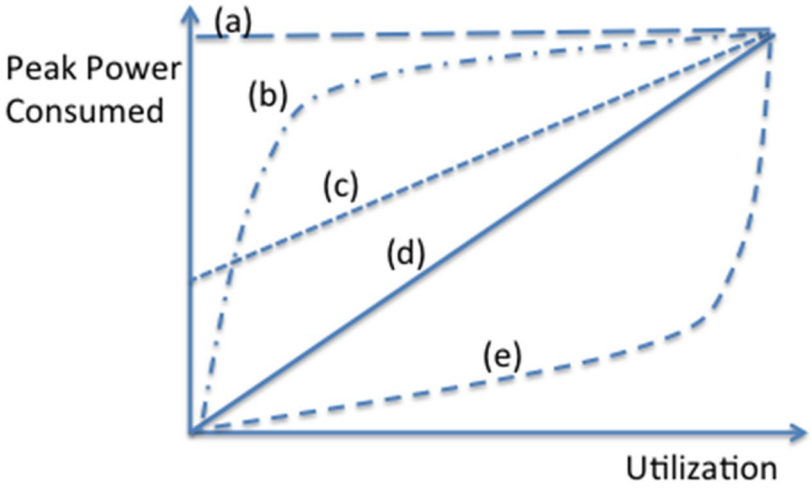
2. Consider the following elements of the datacenter memory hierarchy: memory and disk on a local processing node, a node in the same rack, or a node in a different rack. Which one of the following statements is false?
   (a) Local disk is about 1000 times slower than local memory.
   (b) Memory on another processor in another rack is three times slower than memory on another processor in the same rack.
   (c) Latency to another disk in the same rack is 10% slower than latency to a local disk.
   (d) Bandwidth to memory in the same rack is 10x that of memory in another rack.
   (e) Local disk bandwidth is 10x the bandwidth to a disk on another node in the same rack.

Answers (b)-(e) are true, as given in a lecture slide about Warehouse-scale Computers.

3. The following strategies may affect datacenter performance and availability. Which one is true?
   (a) Data replication improves availability but not performance.
   (b) Partitioning data into smaller fragments and distributing them across numerous machines improves performance and availability.
   (c) Load balancing of requests across numerous servers only improves availability.
   (d) It is not difficult to distinguish between slow servers and broken servers in the datacenter.
   (e) Compression improves performance and availability.

(a) and (c)-(e) are all false. Replication improves both. Load balancing improves performance. Slow and broken are hard to distinguish. Compression doesn't effect availability.

For the next two questions, consider the following figure, showing five possible relationships between processor utilization (x-axis) and peak power consumed (y-axis):



4. Which line best describes the current state of processor power consumption as a function of utilization? **(c)**
5. Which line represents the closest to the ideal relationship between utilization and peak power? **(e)**

4(c) is best because it shows that power does decline with utilization, but doesn't reach 0.
5(e) is ideal because it shows the lowest power at all levels of utilization among the choices.

Section II: *C Programming*

6. What is the value of **s** after the following code?

```
unsigned int t[] = {0,1,2,3,4,5,6,7};
unsigned int s = sizeof(t);
```

(a) The length of the **t** array.
(b) The number of bytes in the t array.
(c) The number of bytes in one unsigned `int`.
(d) The number of bytes in an unsigned `int` pointer.
(e) Nothing; you cannot find the size of an array.

Array types are actually a little more than constant pointers. When you declare an array literal, the compiler knows how long it is (just look at it, it has 8 elements!). This is not true when using array parameters and certainly not true when using pointers. sizeof() gives a size measured in bytes of a type or variable.

7. The **`%s`** format specifier takes in a **`char*`** and prints until it finds a null character. What do the following two lines of code print?

```
char *s = "uncharacteristic";
printf("%s",s+s[7]-s[6]);
```

(a) "uncharacteristic"
(b) "ncharacteristic"
(c) "characteristic"
(d) "haracteristic"

(e) Nothing; the address is not valid

s[7] is 'c' and s[6] is 'a'. 'c'-'a' is 2, because they are numbers, and according the layout of ASCII character tables, the letters are in alphabetical order next to each other. Alternatively, the exact number values are on the back of the green sheet. Printing s+2 prints from the 3rd character to the end of the string.

8. What does the following method do?

```
char * bizarre(char *f,char y) {
   char *h = f;
   for (h=f; *h!=y&&*h;)
       h++;
       if(*h) {
         *h = 0;
          h++;
       }
   return h;
 }
```

(a) It returns a pointer to the first location of **`y`** in **`f`.**

(b) It splits `f` at the first occurrence of `y` and then returns a pointer for the remaining string.

(c) It finds the last location of **`y`** in **`f`**, zeros out that location, and then returns a pointer to the next location.

(d) It zeros out most of **`f`** until it finds **`y`**. It then returns a pointer to the location of **`y`** in **`f`**.

(e) Nothing; it cannot be compiled.

This code loops through the f string until it finds y or the end of the string. Note that the if statement is not inside the loop. If it found y, it replaces it with a null terminator and returns a string starting at the following character. Otherwise, it returns an empty string. The end result is that the f string will stop where y was found, and the returned string will start after that point. This is literally splitting that string. This method clearly zeros something out: only the first occurrence of y. This eliminates (a) (c) and (d). The code compiles, so (e) is also wrong.

9. What does this code print?

```
unsigned char a = 0b01110110;
char b = 0b10011010;
b >>= 1;
b ^= 0b00001101;
unsigned char c = a & b;
printf("0x%02x",c);
```

(a) `0x40`

(b) `0x36`

(c) `0x44`

(d) `0x01`

(e) `0x00`

The operations are an arithmetic right shift, an exclusive or, and an and. Running this code will verify the answer. The %02x means use hex, pad with 0s to 2 characters.

```
Line    Code
0       typedef char * yarn;
1       char * get_word() {
2          return "secret";
3       }
4       void print_word(yarn thread) {
5          printf("%s\n",thread);
6       }
7       int main() {
8          print_word(get_word());
9          return '\0';
10      }
```

(a) Line 0: A **`typedef`** is being used on a pointer.
(b) Line 2: It returns a pointer to local data that will go out of scope.
(c) Line 8: A **`char*`** is given where a **`yarn`** was expected.
(d) Line 9: A null terminator is returned instead of an **`int`**.
(e) Nothing is wrong with this code.

Line 00 is okay; a typedef can be used for pointer types. Line 02 is okay; string literals are part of the static data segment of a program (such as .data in MIPS), and are not locally scoped. Line 08 is okay; this is how a typedef works. Line 09 is okay; '\0' is a char, which is an integer type, and you can put a smaller integer type into a larger integer type without warnings. Thus the answer is (e).

Section III: *C-to-MIPS*

A Stanford student has attempted to convert some C code into MIPS. The original C code was:

```
int add_all_the_nums(int* first_num, int num_count){
  int sum = 0;
  for(int i=0; i < num_count; i++){
      sum += first_num[i];
  }
  return sum;
}
```

The student's solution was:

```
Line    Code
        ADD_ALL_NUMS:
1         add  $t0,$0,$0

        LOOP:
2         beq  $a1,$0,END
3         lw   $t1,0($a0)
4         add  $t0,$t0,$t1
```

```
5         addi $a1,$a1,-1
6         addi $a0,$a0,1
7         j    LOOP

        END:
8         add  $v0,$t0,$0
9         jr $ra
```

As you can probably guess, this MIPS code has problems.

11. Which of the following changes will fix a problem with the code, assuming that the input is always valid?

(a) change line 3 to `lw $t1,4($a0)`
(b) change line 7 to `jal LOOP`
(c) change line 6 to `addi $a0,$a0,4`
(d) change line 8 to `or $v0,$t0,$0`
(e) change line 5 to `slt $a1,$a0,$a1`

Since this method takes in an array of integers, and integers are 32-bits (or 4 bytes), the code to advance the pointer to an integer should be incrementing by 4 instead of by 1, which is answer (c). (a) would incorrectly cause the code to skip the first element. (b) would incorrectly overwrite $ra without first saving it, messing up the return at the end. (d) has no effect, since OR with 0 is basically the same as adding with 0. (e) attempts to compare an array's size with an address, which makes no sense.

For the following two questions, we have partially encoded the instruction at line 2, **`beq $a1,$0,END`** into hex:

Opcode	rs	rt	immediate
000100	?????	?????	???? ???? ???? ????

12. What should go in the **`rs`** field?
(a) 00101
(b) 01001
(c) 00100
(d) 00001
(e) 00000

In the beq instruction, the $rs register is $a1. This translates to register number 5 (which can be found by looking on the Green Sheet). 5 written as a five digit binary number is 00101, and so (a) is the correct answer.

13. What should go in the **`immediate`** field (assuming no delayed branching)?
(a) 1111 1111 1111 1011
(b) 0000 0000 0000 1100
(c) 0000 0000 0000 0110
(d) 0000 0000 0000 0101
(e) 0000 0000 0001 0100

This beq instruction branches to the instruction located at the label END, which is line 8. To encode this information into the 16-bit immediate field, we formulate it as a signed binary number representing the number of instructions to skip relative to the instruction after the beq, which is line 3. We want to jump to line 8, which is five instructions after line 3. So we have to encode 5 as a 16-bit signed immediate, which is (d) 0000 0000 0000 0101.

14. Assume that the code was loaded into the 0x0 segment (the upper four bits of PC are all 0's), and that line 7, **`j LOOP`**, was encoded as follows:

Opcode	address
000010	0b 000 0010 000 0000 0000 0000 0010 (0x100002)

What is the address of line 6, **`addi $a0,$a0,1`**?
(a) 0x00100002
(b) 0x00400008
(c) 0x0040000C
(d) 0x00400018
(e) 0x00100012

The basic idea behind this questions was to use the information provided by the 26-bit address field of the jump instruction, combined with the fact that the question gives that the upper four bits are 0, to reconstruct the address of the instruction at LOOP. To do this we take the 26 bits encoding LOOP, 0x100002, and add two bits to the end (which is the same as multiplying by 4) to get 0x400008. The question gives that the upper four bits of the PC are all 0's, so we add on four zeros to the beginning, so we have 0x00400008 as the address of LOOP (which is pointing to line 2). We know that line 6 is 4 instructions after line 2, which means it is 16 bytes after, so we add 16 to the address. 16 is 0x10 in hex, and so the answer is 0x00400018 (d).

15. In line 5, **`addi $a1,$a1,-1`** there is a negative immediate. Immediates are embedded as 16-bit numbers, but this instruction is trying to add it to a 32-bit number in **`$a1`**. What happens to the 16-bit immediate during the execution of this instruction?
(a) 16 0's get added in front of it.
(b) It gets shifted to the left by 16 bits.
(c) 16 1's get added in front of it.
(d) It gets shifted to the right by 16 bits.
(e) It gets multiplied by 4.

When converting 16-bit immediates to 32-bits for add, they are sign-extended. This means we copy the first bit 16 times. Since this is a negative number, the first bit is 1 so we just add 16 1's, which is (c).

Section IV: *Compilers and Loaders*

For the next three questions, match the phrase that best defines the concept:

(a) Translates assembly language into binary instructions.
(b) Translates source code into intermediates and immediately executes it.
(c) Combines independent programs and resolves labels into an executable file.
(d) Related collection of subroutines and data structures.
(e) Translate a high-level language into assembly language.

16. Compiler (e)

17. Assembler (a)

18. Module (d)

For the next two questions, match the term best associated with the given definition:

(a) Assembly Language
(b) Machine Language
(c) Source Language
(d) Target Language
(e) Object Language

19. Symbolic representation of a computer's binary encoding (a)

20. High-Level Languages such as C or Java (c)

Section V: *Number Representations*

21. Given the following numbers and the representations they should be interpreted as, which of the following is the correct ordering of *least* to *greatest*?
    i. `0xC0700000` in *floating point*
    ii. `0xFFFFFFFC` in *two's complement*
    iii. `0xFF800000` in *floating point*
    (a) i, ii, iii
    (b) iii, ii, i
    (c) iii, i, ii
    (d) i, iii, ii
    (e) ii, i, iii

i, ii, ii turn out to be -3.75, -4, -inf, so ordering them from least to greatest is (b)

22. Flipping all of the bits in ones complement is the same as:
    (a) Adding one to the number.
    (b) Subtracting one from the number.
    (c) Subtracting the number from 2^n-1, where n is the number of bits.
    (d) Dividing the number by negative one.
    (e) Multiplying the number by negative one, then subtracting one.

23. What is the smallest positive integer 32 bit IEEE floating point *cannot* represent?
    (a) 1
    (b) $2^{24} + 1$
    (c) $2^{128} - 1$

(d) $2^{127} + 2^{126} + 2^{125} + \ldots + 2^{104} + 1$
(e) $2^{128} + 2^{127} + 2^{126} + \ldots + 2^{105} + 1$

Because we have limited amount of precision, certain numbers are not representable. The first integer that loses precision due to roundoff is (b), since 1 added to 2^{24} tries to change the bit in the 24th place of the mantissa, we lose this bit and we end up with 2^{24}

24. Give the result of adding the following binary numbers in *two's complement*. Assume we are working with 4 bit binary numbers only: **1010 + 1110**
    (a) -8_{ten}
    (b) 4_{ten}
    (c) -4_{ten}
    (d) Overflow
    (e) Underflow

25. Executing the following C code:

```
#include <stdio.h>
int a = 11;
if (0x2a == 42 && a) {
   printf("The cat is a cat cat.");
}
```

    will:
    (a) Print the sentence
    (b) Error when it gets to the third line
    (c) It doesn't actually compile
    (d) Error when it gets to the fourth line
    (e) None of the above

Although this problem was ambiguous in that it did not specify whether this was the entirety of the source code, the question specifically asks what would happen regarding only the 5 lines of code. The best answer in this case is (a), because the if statement will evaluate to 'true'.

Section VI: Memory Hierarchy

26. Which of the following best describes an action likely to require fewer clock cycles than loading a whole cache line from memory.
    (a) Loading a subset of the cache line from memory.
    (b) Writing to a subset of the cache line with a write-through policy.
    (c) Writing to a subset of the cache line with a write-back policy.
    (d) Accessing a very small portion of the disk.
    (e) None of the above.

Writing to a subset of the cache line with the write-back policy is the one that can require fewer clock cycles than loading cache line from memory because writes can occur without having to access memory. A and B requires a memory access for both, so

they are going to require approximately same amount of clock cycle as the loading the cache line. Accessing the disk would take extremely long time.

27. In the worst case scenario, cache performance can be:
    (a) Worse than a series of direct access to memory.
    (b) Still better than a series of direct access to memory.
    (c) Equivalent to a series of direct access to memory.
    (d) No relevance to a series of direct access to memory.
    (e) Equivalent to not having the cache there.

A cache can have worse performance than having direct access to memory, as it could have a really bad situation in which it needs to access memory each time, and it would have an extra overhead of accessing cache and other computations involved with it.

28. The proportional difference in access time from CPU to L1 cache is _____ that from cache to memory.
    (a) Greater than
    (b) Less than
    (c) The same as
    (d) Undefined in respect to
    (e) Not enough information

Although the clarification might have been a bit confusing, this problem was asking for the proportional difference in access time. That means that since CPU to L1 cache has about 1:1 ratio and cache to memory about 1:100 ratio (can vary a bit depending on how you see it, but within that order of magnitude), the difference would be near 0 for CPU to L1 and near 99 for L1 to memory. Even with the clarification on board about the "ratio", the difference of ratio would still be less for CPU to L1.

29. The write policy of the cache:
    (a) Changes the memory address to which the data is written.
    (b) Causes a memory access to fail.
    (c) Affects the hit rate of read accesses on the cache.
    (d) Must be declared by software.
    (e) None of the above.

The write policy of the cache does not affect where in memory things are written. It also does not cause any memory access to fail or affect the hit rate, as it has no interaction with what gets loaded into cache or memory. Cache write policy is practically never declared by software and is determined by hardware.

30. If instructions without memory-related functions are called:
    (a) There is no way the caches would get accessed.
    (b) There is no way the memory will be accessed.
    (c) There is no way the disk will be accessed.
    (d) All of the above.
    (e) None of the above.

One must recall here that all programs are stored in disk, transferred to memory, then into cache and CPU, regardless of whether they are memory-related functions or not. Therefore, it is none of the above.

Section VII: Cache Performance

Answer the following questions assuming the following memory/cache organization:

4 GiB Byte-Addressed Memory
256 KiB Direct-Mapped Cache
8 Byte Words
4 Word Blocks

31. Give the Tag : Index : Offset breakdown for the above cache:
    (a) 16 : 15 : 4
    (b) 14 : 13 : 3
    (c) 16 : 13 : 3
    (d) 14: 13 : 5
    (e) None of the above

Firstly, we are told that our blocks contain 4 words and each word is 8 Bytes. Therefore, we find that we have 4*8 = 32 Byte blocks. Now, we know that we must be able to "label" each of the 32 bytes inside of this block. In order to "label" 32 items in binary, we need $\log_2$(blocksize) bits. Therefore, we have 5 bits of offset. Next, we must calculate the number of indices (blocks) to get the number of index bits. In order to calculate the number of blocks, we take the size of the cache (256 KiB, or 2^8*2^10 = 2*18 Bytes) and divide it by the previously calculated blocksize (32 = 2^5 bytes). Doing this, we get 2^18/2^5 = 2^13 Indices. To get the number of index bits, we take the log base 2 of this to get a 13 bit Index section of the address. Finally, we take our total address size (log base 2 of the memory size), 32 and subtract the number of Index and Offset bits to get the Tag. This gives us 32 – 13 – 5 = 14. Therefore, our T:I:O breakdown is 14:13:5.

32. Suppose we switch to a word-addressed memory. Give the Tag : Index : Offset breakdown:
    (a) 16 : 15 : 4
    (b) 14 : 13 : 3
    (c) 16 : 13 : 3
    (d) 14: 13 : 5
    (e) None of the above

In a word-addressed memory, we must be able to label every WORD in memory (as opposed to every byte). Therefore, let us find the total address size by taking our memory size in bytes and dividing it by the number of bytes per word. This gives us: (2^32 Bytes / (2^3 bytes per word)) = 2^29 words. Thus, we need 29 bits for our addresses. Next, we know that we must be able to label each word in a block. Each block consists of 4 words, so we'll have 2 bits of offset in our addresses. Next, we note that changing from byte addressing to word addressing does not change the number of blocks in our cache. Thus, from our calculation in question 31, we have 13 index bits. Finally, we determine the number of tag bits by subtracting (13+2) from 29. This gives us a T:I:O breakdown of 14 : 13 : 2. This is not listed as an answer, so we get None of the above.

33. Using the following code to answer the question. You may assume **`sizeof(int) = 4`**. Also assume that **`arr`** is block aligned.

```
#define WIDTH _?_
#define STRIDE _?_
void printer(int* arr) {
   // assume arr is of length WIDTH
   for (int i = STRIDE; i < WIDTH; i += STRIDE) {
        printf("%d\n", *(arr+i));
        printf("%d\n", *(arr+i-STRIDE));
   }
}
```

Suppose WIDTH = 2^{20}. What is the smallest stride that produces the worst possible hit rate and what is that hit rate? (assume a cold cache, answers are STRIDE, Hit Rate).

(a) 8, 0%
(b) 16, 0%
(c) 2^{16}, 0%
(d) 1, 50%
(e) 2^{20}, 10%

The smallest hit rate we can achieve is zero percent, which we can produce by setting stride to 2^16. If we do so, each block we load in from memory will go to the same "slot" in cache (note that with a stride of 2^16, the elements we access successively in the array are 2^16*2^2 bytes apart, which equals the size of the cache). Since we only use one item from each block, we get a hit rate of zero percent. Here's a sample pattern:

Access (arr+i) <= this is stride away from the last element we loaded, so that element (and its block is clobbered in the cache)
Access (arr+i-STRIDE) <= this is stride away from the last element, thus we again clobber the same block in cache.
Repeat for a miss rate of 100% => Hit rate of 0%

34. Calculate AMAT for a machine with the following specs: L1 hits take 3 cycles, L1 local miss rate is 25%. L2 hits take 10 cycles, L2 local hit rate is 60%. L3 hits take 100 cycles, L3 global miss rate is 9%. Main memory accesses take 1000 cycles and all data is available in memory.

(a) 105.5 cycles
(b) 1000 cycles
(c) 24.5 cycles
(d) 25.5 cycles
(e) 278.25 cycles

Here we use our standard AMAT formula, but first we can compute the local miss rate for L3 to simplify our formula:

L3 global miss rate = L3 local * L2 local * L1 local = L3 local*0.4*0.25 = 0.09, so L3 local miss rate = 0.9
Then: AMAT = 3 + 0.25*(10 + 0.4*(100 + 0.9*1000)) = 105.5 cycles

35. Suppose we call **`printer()`** repeatedly with the same arr. How does increasing **`STRIDE`** change the "amount" of spatial and temporal locality with respect to **`arr`**?

(a) Increasing STRIDE decreases temporal locality and increases spatial locality.
(b) Increasing STRIDE increases temporal locality and increases spatial locality.
(c) Increasing STRIDE does not change temporal locality and decreases spatial locality.
(d) Increasing STRIDE increases temporal locality and decreases spatial locality.
(e) Increasing STRIDE increases temporal locality and does not change spatial locality.

As we increase stride to a very large value, we drastically decrease the amount of computation we do inside of our loop and as a result access fewer elements in our array (each of which are farther apart). Since the elements are farther apart, we decrease spatial locality. However, since we are accessing fewer of the elements, it is more likely that they will stay in cache – thus repeated calls to printer() take advantage of temporal locality.

Section VIII: Parallelism

36. Which type of parallel computing architecture is no longer commonly encountered in machines today?
(a) MIMD
(b) MISD
(c) SIMD
(d) SISD
(e) SNSD

As discussed in lecture and section, MISD computers are no longer commonly encountered – MISD is mainly of historical interest.

37. When thousands of users are browsing (but maybe not buying) items on Amazon.com, what kind of parallelism are their servers handling?
(a) EC2
(b) Data-level
(c) Hadoop
(d) Request-level
(e) Strong scaling

By definition, Amazon's servers would be handling thousands of independent requests, in parallel.

38. A given program runs x = 100 floating point operations, 10 of which cannot be parallelized. Assuming the other operations can be perfectly parallelized, what is the speed-up for 9 processors?
(a) 10
(b) 5
(c) 9
(d) 11.1
(e) 0.9

Use Amdahl's law for F = 0.9 (90/100 can be parallelized), S = 9 (9 processors), to compute 5.

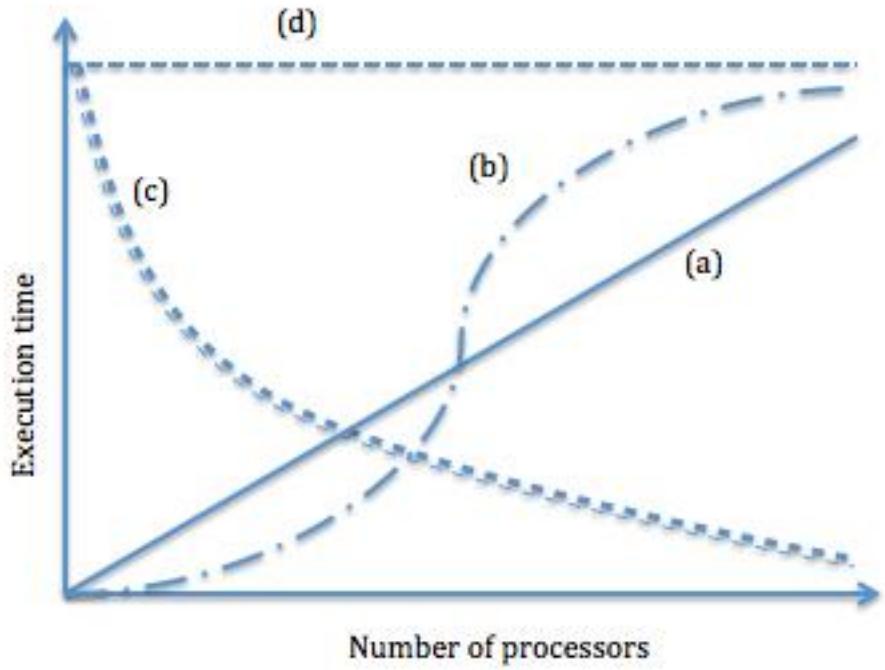
39. You've run two MapReduce jobs using the same map and reduce functions. In the first, you processed 10GB in 7min 22sec, using a 5-machine cluster. In the second, you processed 30GB in approximately the same amount of time, using a 15-machine cluster. What kind of scaling did you observe, if any?
(a) Weak scaling

(b) Strong scaling
(c) Weak and strong scaling
(d) Inverse scaling
(e) No scaling

You only see weak scaling: observing a speedup over your overall problem size while keeping the amount of work per processor constant. Here, with triple the processors we can process three times the amount of data.

40. Assuming a fixed data set, which lines on the graph below best represents strong and weak scaling?
(a) Strong: A, Weak: C
(b) Strong: A, Weak: D
(c) Strong: C, Weak: A
(d) Strong: C, Weak: D
(e) Strong: D, Weak: B

Strong scaling is represented by line C: increasing the number of processors decreases the execution time.
Weak scaling is represented by line D: the line remains constant because the amount of work per processor remains fixed, and so the execution time is unchanged.



Section IX: Lab and Project Knowledge

41. Suppose your Git repository has two branches, `branch1` and `branch2`. `branch1` contains only files `a.txt` and `b.txt`. `branch2` contains only a modified version of `a.txt`, as well as a new file, `c.txt`. If you are in `branch2`, what will happen after you run the command `git merge branch1`?
(a) Nothing, because there's a syntax error with the merge command you just typed.

(b) `Branch1` now contains `a.txt` and `b.txt`. `Branch2` now contains `a.txt`, `b.txt`, and `c.txt`.

(c) `Branch1` now contains `a.txt`, `b.txt`, and `c.txt`. `Branch2` now contains `a.txt` and `b.txt`.

(d) The two branches are merged one branch containing `a.txt`, `b.txt`, and `c.txt`.

(e) The merge does not take place due to the conflicts between the two versions of `a.txt`.

Merging from another branch into the current one updates the current branch, while the other branch doesn't change. A merge between two branches will still occur if there are file conflicts—it just now becomes the job of the user to manually resolve it.

42. Which one of the following statements regarding Hadoop is ***false***?

(a) A combiner in Hadoop must override the reduce() function.

(b) Data types that are used as keys in Hadoop implement the Writable interface.

(c) Multiple copies of a map task may run at the same time.

(d) Hadoop can handle multiple worker failures, but is unable to handle master failure.

(e) A reduce task cannot start until all of its input data has been moved onto the worker.

Part of the reduce task involves getting the data from the mappers, so (E) is false. Furthermore, the reduce task does not *move* the data, but *copies* it. This is an important point that is fundamental to the robustness of Hadoop.

43. You want to change the Map output value type in your Hadoop project from **`IntWritable`** to **`DoubleWritable`**. You've already made all the changes to your **`map()`** and **`reduce()`** functions, including the function signatures, but nothing else. Assume that you are not using a combiner, and that you've included the statement **`import org.apache.hadoop.io.*`** What is the minimum number of additional lines you'll need to change to get your code to compile?

(a) 0

(b) 1

(c) 2

(d) 3

(e) 4

The correct answer is two because we need to change the Mapper output type and Reducer input type needed to implement the respective interfaces (ie. `Mapper<K1, V1, K2, V2>` and `Reducer<K2, V2, K3, V3>`). While you should also make a change in `main()`, as students have pointed out the question asked for the changes required for compilation, and not changing `main()` results in a runtime exception, not a compile-time error.

44. You are debugging the following snippet of code with GDB:

Line	Code
20	`int main(void) {`
21	`  int x = 2, *y = x;`
22	`  *y = square(x);  //square() is defined elsewhere`

You set a breakpoint by typing the command `break 22`, and then you type run. You then type `step`. What line gets printed by GDB when you hit the breakpoint, and what happens after you type `step`?

(a) Line 21, and the program continues execution when you type step.
(b) Line 21, and the program crashes when you type step since line 22 is buggy.
(c) Line 22, and the program continues execution when you type step.
(d) Line 22, and the program crashes when you type step since line 22 is buggy.
(e) Neither lines, the program crashes before GDB can pause at the breakpoint.

Setting a breakpoint at line X stops the program immediately before executing line X, and GDB will print the next line that will be executed (line X). The step command steps into the square function, so line 22 does not get fully executed and therefore the program does not crash yet.

45. Fill in the blank: When MARS encounters a syscall, it checks the ____ register to see which command it should execute.
(a) **`$at`**
(b) **`$a0`**
(c) **`$s0`**
(d) **`$t0`**
(e) `$v0`

Syscalls are defined to check the $v0 register in MARS.

Section X: *MIPS Conventions*

46. The following is pseudocode for a recursive function that finds the total sum of the sizes of each element of a hailstone sequence (you don't need to know what these are).

```
hailstone(int n)
   if(n==1), return n
   else if (n is even), return n + hailstone(n/2)
   else return n + hailstone(3*n+1)
```

Using proper, standard MIPS convention, how many registers should we store to the stack if we were to implement this function with this exact algorithm in MIPS? Assume we use no **`$s`** registers.
(a) 0
(b) 1
(c) 2
(d) 4
(e) None of the above

47. The following is some messy code written by an angsty Stanford student who doesn't want to play by the rules.

```
repeat:
   addi $sp, $sp, -4
   sw $ra, 0($sp)
   beq $a0, $zero, bob
   addi $a0, $a0, -1
   jal task #task is a function written by a Cal student
   lw $ra, 0($sp)
   addi $sp, $sp, 4
   j mary
```

```
bob:
   jr $ra
mary:
   addi $s1,$ra,-4
   jr $s1
```

Assuming I have a program that uses proper MIPS conventions, and assuming task uses proper MIPS conventions, which of the following options could go wrong if I do the following in my program?

```
   addi $a0,$0,8
   jal repeat
```

(i) Memory accesses from the stack in my program will be off.
(ii) **repeat** might run task 1 time instead of 8.
(iii) If **repeat** returns, it might not return to the correct address,

(a) The program will run fine
(b) (i) and (ii)
(c) (ii) and (iii)
(d) (i) and (iii)
(e) (i), (ii), and (iii)

48. The following is a small snippet of code from a function. What is the maximum number of *load* or *store* instructions that can be eliminated from this clip of code so it still works as desired for an arbitrary function passed in **$a1**? Assume the function in **$a1** obeys proper MIPS conventions, and that the clip of code works properly with none of the loads or stores eliminated.

```
# if f is the function stored in $a1,
# then this is supposed to return f($a0) + f($a0)
# code before
   move  $s0,$a1
   addi  $sp,$sp,-16
   sw    $ra,0($sp)
   sw    $a0,4($sp)
   sw    $s0,8($sp)
   jalr  $s0
   sw    $v0,12($sp)
   lw    $a0,4($sp)
   lw    $s0,8($sp)
   jalr  $s0
   lw    $t0,12($sp)
   add   $v0,$v0,$t0
   lw    $ra,0($sp)
```

```
   addi  $sp,$sp,16
#code after
```

(a) less than 2
(b) 2
(c) 3
(d) 4
(e) 5 or more

49. Which of the following is guaranteed to stay unchanged after you call a function that uses good MIPS conventions?
(a) **`$ra`**
(b) **`$a2`**
(c) **`$sp`**
(d) **`$v1`**
(e) More than one of the above

50. Which of the following is true?
(a) MIPS can store words to memory locations occupied by the program (e.g., **`sw $s0, 0($ra)`**)
(b) addiu will error if overflow or underflow occur
(c) MIPS is primarily little-endian
(d) MIPS uses the CISC architecture paradigm
(e) More than one of the above