1. The following code fragment is the inner-loop of a bubble sort. Translate it to SPARC assembly language. You may assume the following: variable `i` is available in register `%l0`, variable `n` is available in register `%l2`, variable `items` is an array of integers defined in the data segment (it can be accessed as label `items`), and function `swap` is available for you to call (it swaps `item[i]` with `item[i+1]`). To get full credit you need **not** optimize your code (for this problem).

```
for (i = 0; i < (n-1); ++i) {
   if (items[i] > items[i+1]) {
      swap(items,i);
   }
}
```

2. Procedure `swap` in the previous question is most likely implemented as a leaf subroutine. Leaf subroutines are easier to implement and more efficient to execute. Explain why. What do leaf subroutines need to ensure regarding register usage? Be precise with respect to `%g`, `%i`, `%o`, and `%l` registers.

3. The following C code fragment

```
count = 0;
total = 0;

while( (value = get_int()) >= 0) {
   total += value;
   ++count;
}
```

compiles into the SPARC assembly language given below. (Note: register `%l0` holds the value of `count`, register `%l1` holds the value of `total`, and register `%o0` holds the return value from the function, and thus holds `value`.) Write SPARC assembly language code that has the same semantics as the given code, but is optimized by removing *as many* `nop`'s *as possible*.

```
#define count %l0
#define total %l1
#define value %o0
   .
   .
   .
   clr count
   clr total

   ba test
   nop

loop:
   add total, value, total
   inc count

test:
   call get_int
   nop
   cmp value, 0
   bge loop
   nop
```

4. The following function creates an array of accounts, which are names/balance pairs. Since the function does not know how many accounts it will have to put in the array, the array is adjusted in size whenever it runs out of space. Three helper functions (`am_done`, `get_name`, and `get_balance`) are not shown; assume they all work correctly. Identify and explain the four unique run-time memory errors in this code (one of which occurs twice). (Note: there is nothing wrong with the parameter `number`).

```
struct Account {
  char *name;
  int balance;
};

void get_info(struct Account *current_account) {
  char name_buf[100];

  /* get_name puts name of next account into argument passed it. */
  get_name(name_buf);
  current_account->name = name_buf;
  current_account->balance = get_balance();
}

struct Account *get_accounts(int *number) {
  int i, old_array_size;
  int array_size = 10;
  struct Account *accounts, *current_account, *old_accounts;

  /* accounts should hold the whole array, and current_account
     should point to the element we are currently working on */
  *number = 0; /* Initalize count to 0 */
  accounts = (struct Account *) malloc(10 * sizeof(accounts));
  current_account = accounts;
  while (! am_done()) {
    if (*number == array_size) {
      old_array_size = array_size;
      old_accounts = accounts;
      array_size = array_size * 2;
      accounts = (struct Account *) malloc(array_size * sizeof(accounts));
      for (i = 0; i < old_array_size; ++i)
         accounts[i] = old_accounts[i];
    }
    get_info(current_account);
    ++current_account;
    ++(*number);
  } /* end while loop */
  return accounts;
}
```

5. Fill in the boxes (below and on next page) showing the text and data sections that the `as` assembler will produce when given this SPARC assembly language program. Flag any instructions that cannot be assembled, filling the unresolvable bits with zeros. Express your answer in hexadecimal. The first instruction of each section has been translated for you. Make sure you annotate your answer with explanations in the manner shown below; explanations will be critical for determining partial credit.

```
    .section ".data"
    .ascii "abc"
    .skip 4
    .asciz "b"
    .byte 4
    .align 4
a:  .word 4

   .section ".text"
   add   %r1, %r2, %r3
   sub   %r1, %r2, %r3
   add   %r1, 4, %r3
   add   %r1, 4 + 4, %r3
   sethi %hi(a), %r3
   bg    mylabel
   call  printf
   ld    [%r1 + %r2], %r3
   ld    [%r1 + 4], %r3
mylabel:
   ld    [%r1], %r3
```

Text Section

Offset	Contents (hex)	Explanation
0	86 00 40 02	10 00011 000000 00001 0 00000000 00010
4		
8		
12		
16		
20		
24		
28		
32		
36		
40		
44		

Data Section

Offset	Contents (hex)	Explanation
0	61	ASCII code for ’a’
1	62	ASCII code for ’b’
2	63	ASCII code for ’c’
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		

6. A processor’s instruction set is like a software module’s interface: you want it to stay fixed even if the underlying implementation changes. This allows codes written for early versions of the processor to run on later versions. One way that a processor implementation might change is to add more registers, for example, giving each process access to 64 registers instead of just 32. Unfortunately, this particular implementation change would force a change to the instruction set. Explain why.

7. Answer the questions below for the following C program. Assume all local variables are saved on the stack. Partial credit depends on showing your work.

```
void f (int a0, int a1, int a2, int a3, int a4, int a5, int a6)
{
  int b;

  b = a6;
  if (b > 0)
    f (a0, a1, a2, a3, a4, a5, b-1);
}

int main ()
{
  f(0,0,0,0,0,0,3);
  return (0);
}
```

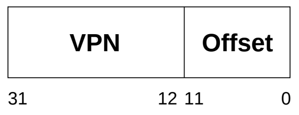
   (a) What is the minimal number of bytes that must be allocated to hold `main`'s stack frame?

   (b) Give the very first assembly language statement of function `f`. Hint: it's a `save` instruction.

   (c) Keeping in mind that the stack changes in size as the program runs, how many total bytes are allocated on the stack when it is at its largest. Ignore any stack space allocated for wrapper functions; just consider the code shown above.

   (d) Give the sequence of assembly language statements that implements statement "`b = a6;`".

8. The MIPS architecture, which today runs in Silicon Graphics computers, was developed at the same time as the SPARC architecture; the two architectures are similar in many respects. One of the differences is how the MIPS architecture partitions the 32-bit address space. The format for a MIPS address is defined as follows:

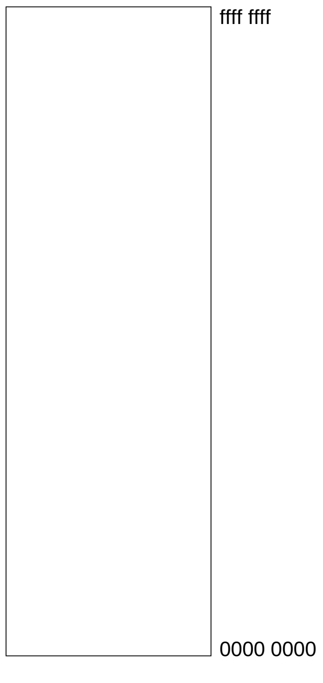


That is, the 32-bit addess is partitioned into a 20-bit VPN (Virtual Page Number) and a 12-bit Offset. This means that memory is divided into 2^{20} pages, each of which is 2^{12} = 4KBytes big. What's unique about the MIPS architecture is how these 2^{20} memory pages are further divided into those that can be accessed by user programs (it's where the text, data, bss, and stack sections are stored), and those that can be accessed by the just the kernel (i.e., only when the processor is running in supervisor mode). Specifically, MIPS memory is further deliniated into four segments based on the first few bits of each address:

user_seg: the most-significant bit of the address (bit 31) is `0`.
kern_seg0: the most-significant three bits of the address are `100`.
kern_seg1: the most-significant three bits of the address are `101`.
kern_seg2: the most-significant two bits of the address are `11`.

It is not important to this question that you understand how the kernel makes use of its three segments (**kern_seg0**, **kern_seg1**, and **kern_seg2**); it is only relevent that there are three kernel segments.

Complete the following schematic of memory by showing where each segment resides in the address space, that is, by giving the addresses (in hex) where each segment begins/ends. Also report the size (in bytes) of each segment.


ffff ffff
0000 0000