**Problem 1** (20 points) **Optimization**

Perform at least **five** standard compiler optimizations on the following C code fragment by writing the optimized version (in C) to the right. *Assume* ***cube, g,*** *and* ***h*** *are pure functions that each return an integer with no side effects to other data structures.*

| Original | | Optimized |
|---|---|---|
| `int cube (int n) {` | | `// same` |
| `  return (n*n*n);}` | | `// same` |
| `int g (int k) { … }` | | `// same` |
| `int h (int i, int j) { … }` | | `// same` |
| `int slowcode(int a, int b) {` | | `// same` |
| `  int t = 100;` | | `int t = 100; int tmp;` |
| `  int p = 1, s=0;` | | `int p = 1;` |
| `  do {` | ⇨ | `do {` |
| `      if (s)` | | |
| `        p += t*cube(p);` | | |
| `      else` | | `    tmp = a*t;` |
| `        p += h(s+a*t, p+a*t);` | | `    p += h(tmp, p+tmp);` |
| `      printf("t:%d, p:%d\n",t,p);` | | `// same` |
| `      t++;` | | `// same` |
| `  } while(t<g(p/256) + h(a, b));` | | `} while(t<g(p>>8) + h(a, b));` |
| `  return h(p, t);` | | `return h(p, t);` |
| `}` | | `}` |

Briefly describe which standard compiler optimizations you applied <u>*and how they improve storage and/or execution efficiency in this code example*</u> (be specific; e.g., "replaces 2 MIPS operations with 1 operation on every loop iteration").

1. **constant propagation (s=0): reduce storage required and operations to allocate/deallocate it as a local.**
2. **dead code elimination: "if (0)" – always zero, so no need for if or then clause, reduces code length and eliminates branch operation**
3. **algebraic simplification: (s+a*t = 0+a*t = a*t): reduces number of operations**
4. **strength reduction (p/256 = p>>8): turns division into a simpler shift operation**
5. **common subexpression elimination (a*t): the subexpression is only computed once.**

**Problem 2** (2 parts, 40 points) **MIPS and C Programming**

**Part A** (16 points) Given two arrays A and B, of 64 integers, write a C fragment that loops through the two arrays and computes the average difference of the corresponding elements (i.e., `A[i]-B[i]` for `i` from 0 to 63) and assigns the result to the variable `avgDiff`. The C fragment should also compute the minimum and maximum of the differences and assign them to the variables `minDiff` and `maxDiff`, respectively. ***For maximum credit, declare and initialize any necessary variables.*** ***NOTE****: A and B can be any 64-element integer arrays, not just the example given. Pay careful attention to how you initialize* `minDiff` *and* `maxDiff` *– the minimum difference could be greater than 0 and the maximum difference might be less than 0.*

```
int   A[64] = {-17, 2, 93, 9, ... -14, 7}; // given
int   B[64] = {-19, 5, 93, 7, ... -14, 8}; // given
int   Diff = A[0]-B[0];
int   avgDiff = Diff;
int   minDiff = Diff;
int   maxDiff = Diff;
int   i;
for (i = 1; i<64; i++){
  Diff = A[i]-B[i];
  avgDiff += Diff;
  if (Diff < minDiff)
    minDiff = Diff;
  if (Diff > maxDiff)
    maxDiff = Diff;
}
avgDiff = avgDiff>>6;
```

**Part B** (24 points) Write MIPS code for the fragment in Part A. **Store the `avgDiff` computed in register $4, the `minDiff` in register $5, and `maxDiff` in register $6**. *For maximum credit use a minimum number of instructions.*

| Label | Instruction | Comment |
|---|---|---|
| | .data | # |
| A: | .word -17, 2, 93, 9, ... -14, 7 | # given |
| B: | .word -19, 5, 93, 7, ... -14, 8 | # given |
| | .text | # |
| | addi $1, $0, 4 | # initialize i ($1) |
| | lw $7, A($0) | # read A[0] |
| | lw $8, B($0) | # read B[0] |
| | sub $9, $7, $8 | # Diff = A[0] - B[0] |
| | addi $4, $9, 0 | # init avgDiff = Diff |
| | addi $5, $9, 0 | # init minDiff = Diff |
| | addi $6, $9, 0 | # init maxDiff = Diff |
| Loop: | slti $10, $1, 256 | # is i < 64 ($1<256)? |
| | beq $10, $0, Exit | # if not, exit loop |
| | lw $7, A($1) | # read A[i] |
| | lw $8, B($1) | # read B[i] |
| | sub $9, $7, $8 | # Diff = A[i] - B[i] |
| | add $4, $4, $9 | # running sum of Diff |
| | slt $10, $9, $5 | # is Diff < minDiff? |
| | beq $10, $0, Skip | # if not, Skip |
| | addi $5, $9, 0 | # update minDiff = Diff |
| Skip: | slt $10, $6, $9 | # is Diff > maxDiff? |
| | beq $10, $0, Skip2 | # if not, Skip2 |
| | addi $6, $9, 0 | # update maxDiff = Diff |
| Skip2: | addi $1, $1 4 | # i++ |
| | j Loop | # keep looping |
| Exit: | sra $4, $4, 6 | # avgDiff = avgDiff/64 |
| | | # |
| | | # |
| | | # |
| | | # |
| | | # |
| | | # |
| | | # |
| | | # |
| | | # |
| | | # |
| | | # |
| | | # |
| | | # |
| | | # |
| | | # |
| | | # |

**Problem 3** (4 parts, 20 points) **Short Answer**

**Part A** (4 points) Write a **single** MIPS instruction that is equivalent to the original fragment. Assume *little endian* byte ordering.

| Original: | Equivalent MIPS statement: |
|---|---|
| `lui $4, 0xFF00` | `lbu $3, 1003($0)` |
| `lw $3, 1000($0)` | |
| `and $3, $3, $4` | |
| `srl $3, $3, 24` | |

**Part B** (4 points) Suppose the instruction "`jal Foo`" is at instruction memory address 2020 and `Foo` is a label of an instruction at memory address 4040. When this instruction is executed, what changes occur to the registers. List all registers that are changed (both general purpose and special purpose) and give their new values.

| Register | Value |
|---|---|
| **$31** | **2024** |
| **PC** | **4040** |

**Part C** (6 points) For each of the following, write a single MIPS instruction to implement the C fragment? Assume variables `A, B, C,` and `D` are of type `int` and are stored in registers `$1, $2, $3,` and `$4`.

| | |
|---|---|
| **`A = B & 7;`** | **`andi $1, $2, 7`** |
| **`C = D / 256;`** | **`sra $3, $4, 8`** |

**Part D** (6 points) Consider the MIPS code on the left which implements the array declaration and access on the right, where the variables **`Z`**, **`Y`**, **`X`**, and **`Value`** reside in `$4`, `$5`, `$6`, and `$7` respectively.

```
addi $1, $0, 48
mult $1, $4
mflo $1
sll $2, $5, 4
add $1, $1, $2
add $1, $1, $6
sll $1, $1, 2
sw $7, Array($1)
```

```
int Z, Y, X, Value;
…

int Array[___?_____][___3_____][___16____];
…
Array[Z][Y][X] = Value;
```

What does this code reveal about the dimensions of Array? Fill in the blanks in the array declaration with the size of each dimension that can be determined from the code. If a dimension cannot be known from this code, put a "?" in its blank. Assume a 32-bit operating system.

**Problem 4** (4 parts, 25 points) **Garbage Collection**

Below is a snapshot of heap storage. Values that are pointers are denoted with a "$". The heap pointer is **$6188**. The heap has been allocated contiguously beginning at **$6000**, with no gaps between objects.

| addr | value | addr | value | addr | value | addr | value | addr | value | addr | value |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 6000 | 8 | 6032 | 12 | 6064 | 16 | 6096 | 12 | 6128 | 8 | 6160 | 8 |
| 6004 | 33 | 6036 | 28 | 6068 | 4 | 6100 | $6004 | 6132 | 60 | 6164 | 0 |
| 6008 | 40 | 6040 | 12 | 6072 | 55 | 6104 | $6016 | 6136 | 75 | 6168 | 16 |
| 6012 | 12 | 6044 | $6016 | 6076 | 8 | 6108 | $6176 | 6140 | 16 | 6172 | 12 |
| 6016 | 0 | 6048 | $6100 | 6080 | 6148 | 6112 | 12 | 6144 | 20 | 6176 | $6132 |
| 6020 | $6100 | 6052 | $6116 | 6084 | 8 | 6116 | $6032 | 6148 | 6046 | 6180 | $6100 |
| 6024 | $6088 | 6056 | 4 | 6088 | 4 | 6120 | $6176 | 6152 | 80 | 6184 | $6116 |
| 6028 | 8 | 6060 | 0 | 6092 | 40 | 6124 | 0 | 6156 | 26 | 6188 | 0 |

**Part A** (9 points) Suppose the stack holds a local variable whose value is the memory address **$6044**. No other registers or static variables currently hold heap memory addresses. List the addresses of all objects in the heap that are *not* garbage.

**Addresses of Non-Garbage Objects**: **6044, 6016, 6088, 6100, 6004, 6176, 6132, 6116, 6032**

**Part B** (6 points) If a reference counting garbage collection strategy is being used, what would be the reference count of the objects at the following addresses?

**Reference count of object at $6044 =** 1

**Reference count of object at $6100 =** 3

**Reference count of object at $6116 =** 2

**Part C** (6 points) If the local variable whose value is the address **$6044** is popped from the stack, which addresses from Part A will be reclaimed by each of the following strategies? If none, write "none."

| | |
|---|---|
| **Reference Counting:** | **6044** |
| **Mark and Sweep:** | **6044, 6016, 6088, 6100, 6004, 6176, 6132, 6116, 6032** |

**Part D** (4 points) What benefit does reference counting garbage collection provide that mark and sweep garbage collection strategy does not provide?

**Benefit:** **It is incremental, no need to stop and collect. It is not cache-hostile. It is simple and more efficient.**

**Problem 5** (2 parts, 30 points) **Doubly Linked Lists**

Consider a doubly linked list that is implemented using the following struct definitions.
***NOTE:*** These are the same as the structs used in Project 2-1, except the `data` field in `llnode_t` is of type `int` and the `DLinkedList` has no `size` field.

```
typedef struct llnode_t {
    int             data;
    struct llnode_t* previous;
    struct llnode_t* next;
}LLNode;

typedef struct dll_t {
    struct llnode_t* head;
    struct llnode_t* tail;
    struct llnode_t* current;
} DLinkedList;
```

**Part A** (12 points) Assume a **32-bit system** and consider the following `create_dlinkedlst` function:

```
DLinkedList* create_dlinkedlist() {
    DLinkedList* newList = (DLinkedList*)malloc(sizeof(DLinkedList));
    newList->head = NULL;
    newList->tail = NULL;
    newList->current = NULL;
    return newList;
}
```

**A.1** What integer is passed to `malloc` when this function executes? **12**.

**A.2** Which region of memory holds the variable `newList`? **stack**.

**A.3** How much space (in bytes) is allocated for `newList` in this region of memory? **4 bytes**.

**A.4** How much space (in bytes) is allocated for the return value of `create_dlinkedlst`?

**4 bytes**.

**Part B** (18 points) Complete the C function **`Insert_Node_After`** that takes a pointer to an `LLNode` and inserts it **after** the `current` node in the doubly linked list pointed to by the input parameter `DLL`. Return 0 if the current node is `NULL`, otherwise return 1 (this code is already provided). Be sure to update the tail of `DLL` if `N` becomes the new tail. DLL's `current` field should not change.

```
int Insert_Node_After (LLNode *N, DLinkedList *DLL) {
  if(DLL->current == NULL){
    return 0;
  }else{
    LLNode *C = DLL->current;
    N->previous = C;
    N->next = C->next;
    if (C == DLL->tail)
       DLL->tail = N;
    else
       (C->next)->previous = N;
    C->next = N;

    return 1;
  }
}
```

**Problem 6** (2 parts, 40 points) Activation Frames

Consider the following C code fragment:

```
typedef struct {
   int Start;
   int End;
} trip_info_t;

int TripAdvisor() {
   int         odometer = 981005;
   int         Gallons[] = {16, 6};
   trip_info_t TI;
   int         rate;
   int         Update(trip_info_t, int [], int *);
   TI.Start    = 180;
   TI.End      = 420;
   rate = Update(TI, Gallons, &odometer);
   return(odometer);
}

int Update(trip_info_t Trip, int G[], int *OD) {
   int         miles, MPG;
   miles       = Trip.End - Trip.Start;
   MPG         = miles/G[1];
   *OD        += miles;
   return(MPG);
}
```

**Part A** (18 points) Suppose `TripAdvisor` has been called so that the state of the stack is as shown below. Describe the state of the stack just before `Update` deallocates locals and returns to `TripAdvisor`. Fill in the unshaded boxes to show `TripAdvisor's` and `Update's` activation frames. Include a symbolic description and the actual value (in decimal) if known. For return addresses, show only the symbolic description; do not include a value. *Label the frame pointer and stack pointer.*

| | address | description | Value |
|---|---|---|---|
| | 9900 | RA of TA's caller | |
| | 9896 | FP of TA's caller | |
| SP, TripAdvisor's FP | 9892 | RV | |
| | 9888 | odometer | 981245 |
| | 9884 | Gallons[1] | 6 |
| | 9880 | Gallons[0] | 16 |
| | 9876 | TI.End | 420 |
| | 9872 | TI.Start | 180 |
| | 9868 | rate | |
| | 9864 | RA | |
| | 9860 | FP | 9892 |
| | 9856 | Trip.End | 420 |
| | 9852 | Trip.Start | 180 |
| | 9848 | G | 9880 |
| | 9844 | OD | 9888 |
| | 9840 | RV | |
| FP: 9840 | 9836 | miles | 240 |
| SP: 9832 | 9832 | MPG | 40 |

**Part B** (22 points) Write MIPS code fragments to implement the subroutine `Update` by following the steps below. *Do not use absolute addresses in your code; instead, access variables relative to the frame pointer.* Assume no parameters are present in registers (i.e., access all parameters from `Update`'s activation frame). You may not need to use all the blank lines provided.

First, write code to properly set `Update`'s frame pointer and to allocate space for `Update`'s local variables and initialize them if necessary.

| label | instruction | Comment |
|---|---|---|
| `Update:` | `add $30, $29, $0` | `# set FP` |
| | `addi $29, $29, -8` | `# allocate locals` |

`# miles = Trip.End – Trip.Start;`

| label | instruction | Comment |
|---|---|---|
| | `lw $1, 12($30)` | `# read T.Start` |
| | `lw $2, 16($30)` | `# read T.End` |
| | `sub $1, $2, $1` | `# T.End-T.Start` |
| | `sw $1, -4($30)` | `# store in miles` |

`# MPG = miles/G[1];`

| label | instruction | Comment |
|---|---|---|
| | `lw $2, 8($30)` | `# read G (base address)` |
| | `lw $2, 4($2)` | `# read G[1]` |
| | `div $1, $2` | `# miles/G[1]` |
| | `mflo $3` | `# result in $3` |
| | `sw $3, -8($30)` | `# store in MPG` |

`# *OD += miles;`

| label | instruction | Comment |
|---|---|---|
| | `lw $4, 4($30)` | `# read OD (address)` |
| | `lw $2, 0($4)` | `# dereference it` |
| | `add $5, $2, $1` | `# *OD + miles` |
| | `sw $5, 0($4)` | `# *OD = *OD+miles` |

`# return(MPG);  (store return value, deallocate locals, and return)`

| label | instruction | Comment |
|---|---|---|
| | `sw $3, 0($30)` | `# put MPG in RV slot` |
| | `add $29, $30, $0` | `# deallocate locals` |
| | `jr $31` | `# return to caller` |