ECE 2035 A	Programming Hw/Sw Systems	Fall 2015
5 problems, 8 pages	Final Exam Solutions	9 December 2015

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* ***`square`****,* ***`f`****, and* ***`g`*** *are pure functions that each return an integer with no side effects to other data structures.*

```
int square (int n) {
  return (n*n);}
int f (int m) { … }
int g (int c, int d) { … }
int mycode(int a, int b) {
  int x = 100;
  int y = 1, z=64;
  do {
      if (z)
        y += x*square(z);
      else
        y += g(z+a*x, y+a*x);
      printf("x:%d,y:%d\n",x,y);
      x--;
  } while(x>f(y*1024)+g(a, b));
  return g(y, x);
}
```

```
int mycode(int a, int b) {
  int x = 100;
  int y = 1;
  int tmp = g(a, b);
  do {
   y += x<<12;
   printf("x:%d,y:%d\n",x,y);
   x--;
  } while(x>f(y<<10) + tmp);
  return g(y, x);
}
```

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

1. **constant propagation (z=64): reduce storage required and operations to allocate/deallocate it as a local.**
2. **dead code elimination: "if (64)" – always nonzero, so no need for if or else clause, reduces code length and eliminates branch operation**
3. **function inlining of square: eliminates overhead of function call and enables additional strength reduction optimization below (y +=x*64*64)**
4. **strength reduction (y +=x<<12; … `f(y<<10)`): turns multiplication into a simpler shift operation**
5. **loop invariant removal (`g(a, b)`): this computation, including the overhead of subroutine call (activation frame allocation/deallocation), is only done once, instead of on every loop iteration.**

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

Suppose we have the following definition which is used to create singly linked lists.

```
typedef struct Link {
   int         ID;
   int         Value;
   struct Link *Next;
} Link;
```

Part A (6 points) Complete the following subroutine which inserts a `Link` (pointed to by the input parameter `NewLink`) into the list just after the `Link` pointed to by the input parameter `Before`. You may assume that neither input parameter is `NULL`. `Before's Next` field may point to another `Link` or it may be `NULL`. `NewLink's Next` field is `NULL`.

```
void SpliceIn(Link *NewLink, Link *Before){
  NewLink->Next = Before->Next ; /* part A*/
  Before->Next = NewLink ; /* part A*/
}
```

Part B Complete the following recursive subroutine which takes a pointer to the head of a linked list and returns a pointer to a copy of the linked list. Follow the steps specified below.

```
Link * CopyList(Link *Head) {
  if (Head == NULL) return NULL ; /* part B.1 */
  Link *LinkCopy ; /* part B.2 */
  LinkCopy = (Link *) malloc (sizeof(Link)) ; /* part B.3 */
  if ( LinkCopy == NULL ){ /* part B.4 */
     printf("Error: Insufficient space.");
     exit(1);
  }
  LinkCopy->ID = Head->ID ; /* part B.5 */
  LinkCopy->Value = Head->Value ; /* part B.5 */
  LinkCopy->Next = CopyList(Head->Next) ; /* part B.6 */
  return LinkCopy;
}
```

Part B.1 (3 points) Fill in what should be returned if the list is empty.

Part B.2 (3 points) Add a local variable called `LinkCopy` that is a pointer to a `Link` object.

Part B.3 (5 points) Allocate space for a `Link` structure using `malloc` and make `LinkCopy` point to the object allocated. Be sure to include appropriate type casting to avoid type errors.

Part B.4 (3 points) Fill in the test for whether `malloc` found enough space which controls the print statement.

Part B.5 (5 points) Copy the values of `Head`'s `ID` and `Value` fields to `LinkCopy`.

Part B.6 (5 points) Call `CopyList` recursively to copy the rest of the list and assign the result to `LinkCopy's Next` field.

Problem 3 (3 parts, 25 points) **Associative Sets**

Consider a hash table that is implemented using the following struct definitions.

```
#define NUMBUCKETS   5
typedef struct Entry {
   int           Key;
   int           Value;
   struct Entry  *Next;
} Entry;

typedef struct {
   Entry         *Buckets[NUMBUCKETS];
   int           Size;
} HashTable;
```

Part A (6 points) What is the value of each of these (assume a 32 bit system):

sizeof(Entry) = **12**	sizeof(HashTable) = **24**

Part B (10 points) Suppose the entries are maintained in a ***sorted*** linked in each bucket in order from small to large keys. Complete the C function `Find_Key` that *efficiently* searches the hash table for an entry corresponding to a specified key (i.e., *it should end the search as early as possible*). It should return a pointer to the matching `Entry` if `Key` is found or return `NULL` if `Key` is not found in the hash table.

```
Entry *Find_Key(HashTable *HT, int Key) {
   Entry    *ThisEntry;
   int      Index;
   int      Hash(int Key); /* function prototype for hash function */

   Index = Hash(Key);
   ThisEntry = HT->Buckets[Index];
   while((ThisEntry != NULL) && (ThisEntry->Key <= Key)){
     if (ThisEntry->Key == Key)
       return(ThisEntry);
     else ThisEntry = ThisEntry->Next;
   }
   return NULL;
}
```

Part C (9 points) Suppose a hash table created using the structs above contains **155** entries total and the entries are evenly distributed across the **5** hash table buckets, each implemented as a **sorted** linked list of `Entry` structs. An application performs **500** lookups of various keys: **375** of the lookups find the key in the list and **125** lookups fail to find the key. The keys that are found are distributed throughout the list so that each position is equally likely to be where a key is found. What is the average number of key comparisons that would be needed for a lookup in this list implementation? (Show work. Note: you may not have to use all data provided.)

L=155/5 = 31

Avg # comparisons = (31+1)/2 = 16

Average number of comparisons: **16**

Problem 4 (3 parts, 20 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	20	6096	16	6128	12	6160	$6004
6004	33	6036	28	6068	4	6100	$6052	6132	$6172	6164	0
6008	0	6040	$6120	6072	$6132	6104	6010	6136	$6016	6168	16
6012	16	6044	80	6076	8	6108	5	6140	72	6172	$6052
6016	$6100	6048	16	6080	$6148	6112	148	6144	20	6176	0
6020	$6172	6052	0	6084	$6172	6116	8	6148	6046	6180	$6004
6024	25	6056	100	6088	4	6120	32	6152	$6080	6184	0
6028	30	6060	0	6092	$6080	6124	$6080	6156	26	6188	0

Part A (9 points) Suppose the stack holds a local variable whose value is the memory address **$6120** and register $3 holds memory address **$6016**. No other registers or static variables currently hold heap memory addresses. List the addresses of all objects in the heap that would be marked by a mark-and-sweep garbage collection algorithm.

Addresses of Marked Objects: **6120, 6080, 6148, 6004, 6172, 6052, 6016, 6100**

Part B (3 points) If a reference counting garbage collection strategy is being used, what would be the reference count of the object at address **$6172**?

Reference count of object at $6172 = 2

Part C (8 points) If the local variable whose value is the address **$6120** is popped from the stack, which addresses will be reclaimed by each of the following strategies? If none, write "none."

Reference Counting:	**6120**
Mark and Sweep:	**6120, 6080, 6148**
Old-New Space (copying):	**6120, 6080, 6148**

Problem 5 (3 parts, 30 points) **MIPS and C programming**

Part A (8 points) Suppose the instruction "`jal Foo`" is executed which changes the values of the following registers to:

Register	Value
$31	3216
PC	3620

What is the address of the first instruction of the subroutine `Foo` and what is the address of the `jal Foo` instruction?

Subroutine `Foo` starts at address: 3620

Address of `jal Foo` instruction: 3212

Part B (12 points) Suppose variables `A, B,` and `C` are of type `int` and are stored in registers `$1, $2,` and `$3`. Write a MIPS code fragment that computes `C = A * min(A, B);`. Use only registers `$0, $1, $2,` and `$3` and for maximum credit, use a minimal number of instructions and include comments.

Label	Instruction	Comment
	slt $3, $1, $2	**# is A<B?**
	bne $3, $0, Min	**# if so, branch to Min**
	mult $1, $2	**# if not, compute A*B**
	j End	**# and end**
Min:	**mult $1, $1**	**# Min: A*A**
End:	**mflo $3**	**# put result in $3**

Part C (10 points) What does the following code fragment print?

```
int A[10] = {1990, 3, 1992, 5, 1999, 1, 2001, 3, 2005, 1};
int B[10] = {1991, 2, 1993, 1, 1998, 7, 2002, 1, 2007, 6};
int i,j;
for(i=1; i<10; i=i+2)
  {
    for (j=1; j<10; j=j+2)
      {
        if (A[i] == B[j])
          {
            printf("C: %d, As: %d, Bs: %d.\n", A[i], A[i-1], B[j-1]);
            break;
           }
      }
  }
```

```
C: 1, As: 1999, Bs: 1993
C: 1, As: 2005, Bs: 1993
```

Problem 6 (40 points) Activation Frames

The function `Bar` (below left) calls function `Foo` after completing `code block 1`. Write MIPS code that properly calls `Foo`. Include all instructions between `code block 1` and `code block 2`. **Note that `code block 1` may change the values of the local variables** (e.g., assume `i` can be any value from 0 to 2). Symbolically label all required stack entries and give their initial values if known (below right).

```
int Bar() {
   int        A[] = {9, 25, 27};
   int        i =  0;
   int        y = 5;

   (code block 1)

   A[1] = Foo(A, &y, A[i]);

   (code block 2)

}
```

Bar's FP 9900	XXX	XXX
9896	A[2]	27
9892	A[1]	25
9888	A[0]	9
9884	i	0
SP 9880	y	5
9876	Bar's RA	N/A
9872	Bar's FP	9900
9868	A	9888
9864	&y	9880
9860	A[i]	A[i]
9856		

label	instruction	comment
	addi $29, $29, -24	# allocate activation frame
	sw $31, 20($29)	# preserve bookkeeping info
	sw $30, 16($29)	
	addi $1, $30, -12	# compute A
	sw $1, 12($29)	# push A
	addi $2, $30, -20	# compute &y
	sw $2, 8($29)	# push &y
	lw $2, -16($30)	# compute A[i] by loading i,
	sll $2, $2, 2	# scaling it, and adding it to
	add $2, $2, $1	# the base address of A
	lw $3, 0($2)	# push A[i]
	sw $3, 4($29)	
	jal Foo	# call Foo
	lw $31, 20($29)	# restore bookkeeping info
	lw $30, 16($29)	
	lw $2, 0($29)	# read return value
	sw $2, 8($1)	# store return value in A[2]
	addi $29, $29, 24	# deallocate activation frame