ECE 2035	Programming Hw/Sw Systems	Spring 2013
5 problems, 8 pages	Final Exam Solutions	1 May 2013

Problem 1 (3 parts, 30 points) **Compilation, Concurrency & Interrupts**

Part A (20 points) Perform at least **five** standard compiler optimizations on the following C code fragment by writing the optimized version (in C) to the right.

```
int foo(int x, int y) {

  int g = 32;

  int h=0, a = 0, b = 1, i=100;

  while ((x*y)<i) {

    a += g*(h+i);

    b *= exp(a, (h+i));

    i--;

  }

  return (b);

}
```

⇨

```
int foo(int x, int y){

  int a=0,b=1,i=100;

  int tmp = x*y;

  while(tmp<i){

    a += i << 5;

    b *= exp(a, i);

    i--;

  }

  return(b);

}
```

Briefly describe which standard compiler optimizations you applied:

1. **Dead code elimination (declaration of g)**
2. **constant propagation (g=32 and h=0)**
3. **algebraic simplification (h+i = 0+i = i)**
4. **strength reduction (32*i = i<<5)**
5. **loop invariant removal (x*y)**
6. **(also common subexpression elimination: (h+i) can be computed once and simplified)**

Part B (6 points) In the table below, draw lines to match the type of concurrency with the architectural support required for it.

Type of concurrency	**Draw lines here:**	*Architectural Support*
Data-level parallelism (DLP)		Multicore system with shared memory
Instruction-level parallelism (ILP)		Multimedia ISA extensions (e.g., MMX, SSE)
Thread-level parallelism (TLP)		Pipelining

Part C (4 points) What is the difference between a nonmaskable interrupt (NMI) and an asynchronous interrupt?

An NMI always has priority over asynchronous interrupts. It cannot be masked or overridden by a different interrupt. An asynchronous interrupt (e.g., from keystrokes) can be masked by another interrupt (e.g., from the graphics card).

Problem 2 (4 parts, 35 points) **Associative Sets**

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

```
typedef struct Entry {
   int           Key;
   int           Value;
   struct Entry  *Next;
} Entry;

typedef struct {
   Entry         *Buckets[13];
   int           Size;
} HashTable;
```

Part A (5 points) If a hash table of type `HashTable` is created which contains **273** entries (each of type `Entry`), how many total words of storage are used by this hashtable? (Show work.)

273*3 + 13 + 1 = 833

Part B (5 points) Suppose the same `Value`s that are contained in the 273 entries are stored in an indexed set (an array), each `Value` indexed by the entry's `Key`. Assume the `Key` can be any unsigned integer representable by 32 bits. How many total words of storage are used by the indexed set? (Show work.)

2^{32} entries = 4B words

Part C (10 points) Suppose the bucket lists in the `Buckets` array in **Part A** contain a number of `Entry` objects, evenly distributed across the hash table buckets. Assume that *computing the hash function* takes an average of **five operations** and *comparing two keys* takes an average of **four operations**. Ignore effects of spatial and temporal reference locality. Suppose that **75%** of keys looked up are found in the hash table and **25%** are not found. How many of these operations would be required for the average lookup in the hash table described above if the bucket list is unsorted versus sorted? (Show work.)

number of operations when each bucket list is *unsorted:* **5 + (3/4)(22/2)4 + (1/4)(21)4 = 5 + 3(11) + 21 = 59**

number of operations when each bucket list is *sorted:* **5+ 4(21+1)/2=49**

Part D (15 points) Write a C subroutine called `CreateEntry` that takes two integers (named `K` and `V`), allocates an object of type `Entry` on the heap with fields `Key=K`, `Value=V`, and `Next=Null`, and returns a pointer to the `Entry` object allocated. The subroutine should be sure to check whether there was enough space on the heap for the object to be allocated and if not, it should print an error message and exit.

```
Entry *CreateEntry(int K, int V) {
  Entry *NewE = (Entry *)malloc(sizeof(Entry));
  if (NewE == NULL){
     printf("Insufficient memory");
     exit(1);
  NewE->Key = K;
  NewE->Value = V;
  NewE->Next = NULL;
  return(NewE);
  }
```

Problem 3 (1 parts, 25 points) **Heap and Hash Table**

Consider an open hash table composed of a four-bucket table, with each bucket containing a variable length list. Each list entry has three slots <key, value, next> corresponding to the three word groupings in the entries section. The hash function is `key mod four`. Inserted entries are *appended to the end* of a bucket list. Deallocated entries are maintained on a LIFO free list. When the free list is empty, new entry objects are allocated from heap memory. Accesses are listed as <op, key, [value]>. Simulate the access list below and draw the ending state. Assume the hash table is initially empty, the heap pointer is initially 5016 and the free pointer is initially 0.

Heap Pointer	~~5016~~ ~~5028~~ ~~5040~~ ~~5052~~ 5064	Free List	~~0000~~ ~~5016~~ ~~5052~~ 5016

Buckets

5000		5004	5028	5008	5052	5012	~~5016~~ 5040

Entries

5016	2003	5040	2007	5064		5088	
5020	111	5044	~~333~~ 555	5068		5092	
5024	~~0~~ ~~5040~~ 0	5048	0	5072		5096	
5028	2001	5052	~~2005~~ 2002	5076		5100	
5032	222	5056	~~444~~ 666	5080		5104	
5036	~~0~~ ~~5052~~ 0	5060	~~0~~ ~~5016~~ 0	5084		5108	

Hash Table Access Trace

#	op	key	value	#	op	key	value
1	insert	2003	111	5	insert	2007	555
2	insert	2001	222	6	remove	2003	n/a
3	insert	2007	333	7	remove	2005	n/a
4	insert	2005	444	8	insert	2002	666

Problem 4 (4 parts, 30 points) **Garbage Collection, Function Pointers, and MIPS**

Below is a snapshot of heap storage. Values that are pointers are denoted with a "$". The heap pointer is $6168. 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	16	6032	12	6064	0	6096	16	6128	12	6160	0
6004	33	6036	28	6068	4	6100	6172	6132	72	6164	0
6008	$6132	6040	$6120	6072	$6100	6104	16	6136	9	6168	0
6012	16	6044	$6080	6076	8	6108	5	6140	$6004	6172	0
6016	80	6048	16	6080	$6036	6112	$6148	6144	20	6176	0
6020	8	6052	$6092	6084	6012	6116	8	6148	6046	6180	0
6024	25	6056	$6024	6088	4	6120	32	6152	8	6184	0
6028	$6036	6060	0	6092	$6080	6124	$6024	6156	26	6188	0

Part A (6 points) Suppose the stack holds a local variable whose value is the memory address $6052 and register $3 holds the address $6004. 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: **6052, 6092, 6024, 6080, 6036, 6120, 6004, 6132**

Part B (6 points) If the local variable whose value is the address $6052 is popped from the stack, which addresses will be reclaimed by each of the following strategies? If none, write "none."

Reference Counting:	**6052, 6092**
Mark and Sweep:	**6052, 6092, 6024, 6036, 6120, 6080**
Old-New Space (copying):	**6052, 6092, 6024, 6036, 6120, 6080**

Part C (8 points) Complete the C code below by following these two steps:

1. Create a local variable, called `compare`, in `My_Search` that is a function pointer that points to `GT` if `ascending` is nonzero and to `LT` otherwise. Define the function pointer type with `typedef`.
2. Pass this function pointer to the subroutine `Climb` as its third parameter.

```
int GT(int x, int y)
   return(x>y);
int LT(int x, int y)
   return(abs(x)<abs(y));
typedef     int (* FP) (int, int);                  ; /* part 1*/
int My_Search(int a, int b, int ascending) {

   FP compare;                                      ; /* part 1*/

   compare = (ascending? GT : LT)                   ; /* part 1*/

   Climb(a, b,    compare                     );      /* part 2 */
   ...rest of My_Search's body...
}
```

Part D (10 points) Write the MIPS code implementation of the following C program fragment.

```
int A[100] = {4, -1, 3, …, 17};
int B[25];
int i;
for(i=0; i<25; i++)
   B[i] = A[4*i];
```

Modify only registers $1, $2, $3, and $4. *For maximum credit, include comments.*

Label	Instruction	Comment
	.data	
Aaddr:	.word 4, -1, 3, …, 17	# int A[100]={4,-1,3,…,17};
Baddr:	.alloc 25	# int B[25];
	.text	
	addi $1, $0, 0	# initialize loop counter
Loop:	slti $2, $1, 100	# is counter < 100 (i<25)
	beq $2, $0, Exit	# if not, Exit Loop
	sll $3, $1, 2	# 4*i
	lw $2, Aaddr($3)	# read A[4*i]
	sw $2, Baddr($1)	# write it to B[i]
	addi $1, $1, 4	# increment counter
	j Loop	# loop back
Exit:	...	# instructions after the loop

Problem 5 (1 parts, 30 points) Activation Frames

The function `Bar` (below left) calls function `Foo` after completing `code block 1`. Write MIPS assembly code that properly calls `Foo`. Include all instructions between `code block 1` and `code block 2`. Symbolically label all required stack entries and give their values if they are known (below right).

```
int     Bar() {

   int      A[] = {5, 25};
   int      N = 0;

      (code block 1)

      A[0] = Foo(A, A[1], &N)

      (code block 2)
}
```

	address	label	value
Bar's FP	9900	XXX	XXX
	9896	A[1]	25
	9892	A[0]	5
~~SP~~	9888	N	0
	9884	**FP**	**9900**
	9880	**RA**	**N/A**
	9876	**A**	**9892**
	9872	**A[1]**	**25**
	9868	**&N**	**9888**
Foo's FP, SP	9864	**RV**	**N/A**
	9860		
	9856		

label	instruction	comment
	addi $29, $29, -24	# allocate activation frame
	sw $30, 20($29)	# preserve bookkeeping info
	sw $31, 16($29)	
	addi $1, $30, -8	# push inputs
	sw $1, 12($29)	
	lw $1, -4($30)	
	sw $1, 8($29)	
	addi $1, $30, -12	
	sw $1, 4($29)	
	jal Foo	# call Foo
	lw $31, 16($29)	# restore bookkeeping info
	lw $30, 20($29)	
	lw $1, 0($29)	# read return value
	sw $1, -8($30)	# store return value in A[0]
	addi $29, $29, 24	# deallocate activation frame