ECE 2035 | Programming Hw/Sw Systems | Fall 2013

5 problems, 10 pages | Final Exam Solutions | 9 December 2013

Problem 1 (3 parts, 30 points) **Compilation and Optimization**

Part A (15 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 g, int h) {

  int x = 64;

  int sum = 0;

  do {

    sum += f(g-1, h%4, g+(h%4));

    g = x64*g + h;

  } while (g<128);

  return (sum);

}
```

⇨

```
int foo(int g, int h) {

  int tmp = h%4;

  int sum = 0;

  do {

    sum += f(g-1, tmp, g+tmp);

    g = g<<6 + h;

  } while (g<128);

  return (sum);

}
```

Briefly describe which standard compiler optimizations you applied:

1. **constant propagation (x = 64)**
2. **dead code elimination (declaration of x)**
3. **common subexpression elimination (h%4)**
4. **strength reduction (multiply → left shift)**
5. **loop invariant removal (h%4 moved outside loop)**

Part B (10 points) Optimize register usage by writing the following code fragment to the right (in MIPS). The optimized fragment must use the same number of instructions and perform the same operations, but the instructions may be in a different order. The result of the code fragment must be stored in $2. Remember $0 must always contain 0; it must not be overwritten. For maximum credit, the optimized version should use only registers $0, $1, $2, and $3. Partial credit will be given if more than this number of registers is used, as long as the code is still correct and equivalent to the unoptimized code to the left.

```
# Original
sub $6, $0, $3
sub $4, $0, $2
addi $7, $6, -1
addi $5, $4, -1
and $8, $5, $3
and $9, $2, $7
or $2, $8, $9
```

⇨

```
# One Solution:
sub $1, $0, $3
addi $1, $1, -1
and $1, $2, $1
sub $2, $0, $2
addi $2, $2, -1
and $2, $2, $3
or $2, $1, $2
```

```
# Another Solution:
addi $1, $3, 1
sub $1, $0, $1
and $1, $2, $1
addi $2, $2, 1
sub $2, $0, $2
and $2, $2, $3
or $2, $1, $2
```

Part C (5 points) The MIPS code fragment in Part B can be reduced to a single MIPS instruction that performs the equivalent computation on inputs $2 and $3. Complete the instruction below by filling in the appropriate operator. (Hint: try sample values of $2 and $3 to see what is computed for the resulting $2.)

```
xor $2, $2, $3
```

Problem 2 (2 parts, 30 points) **Reverse Compilation**

Part A (15 points) In an unexpected move, Seattle-based ice cream vendor MicroSoftServe, producer of low-cal Fruity Stack PopSicles™, announced its merger with the BigBlueBunny company, maker of the popular Chocolate-T J Watson Bars™. The announcement was met with a flurry of excitement that these products will now be available in all Windows Ate My Lunch Cafes throughout the world.

Unfortunately, some of the control software for manufacturing these frozen treats is quite old and has survived only in MIPS assembly. You have been hired as a consultant to recover the original C source code. For the MIPS below, write a C subroutine that best matches the behavior. For full credit and to unfreeze BigBlueBunny's assets, write high level C; *do not transliterate*. For example, use the appropriate loop construct (`for`, `while`, or `do while`). Assume $1 holds an input variable **`Temp`** of type **`int`** and $2 holds an input variable **`MeltingPt`** of type **`int`**. Use the variable **`I`** of type **`int`** for register $3's value and assume the output **`I`** is returned in $3. $4 holds a constant and $5 is used for temporary, intermediate values. **`ChurnRPMs`** is a label to an address in the static data region of memory.

Label	**Instruction**
`Cool:`	`addi $4, $0, 20`
	`addi $3, $0, 0`
`Loop:`	`slt  $5, $2, $1`
	`beq  $5, $0, Exit`
	`addi $1, $1, -1`
	`div  $1, $4`
	`mfhi $5`
	`sll  $5, $5, 3`
	`sw   $5, ChurnRPMs($3)`
	`addi $3, $3, 4`
	`j    Loop`
`Exit:`	`jr   $31`

```
int Cool(int Temp, int MeltingPt){
  int I=0;
  while(MeltingPt < Temp){
    Temp--;
    ChurnRPMs[I] = (Temp%20)*8;
    I++;
  }
  return(I);
}
```

Part B (15 points) Assuming a **64-bit system**, show how the following global variables map into static memory. Assume they are allocated starting at address 3000 and all data types are **word aligned**. For each variable, draw a box showing its size and position in memory. Label the box with the variable name or element of an array (e.g., Name[0]).

```
int     x = 10;
double  D = 300.6;
float   F = 6.2;
float   *fp = &F;
double  *dp = &D;
```

3000	x
3004	slack
3008	D
3012	
3016	F
3020	slack
3024	fp
3028	
3032	dp
3036	

Problem 3 (5 parts, 35 points) **Pointers and Arrays**

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;
   int           NumBuckets;
} HashTable;
```

Part A (4 points) Write a single C statement to allocate a `HashTable` structure and to declare a variable named **`MyHT`** that is a pointer to the newly allocated `HashTable` structure.

```
HashTable *MyHT = (HashTable *)malloc(sizeof(HashTable));
```

Part B (3 points) Write a single C statement to set the **`NumBuckets`** field to 5 in the `HashTable` structure pointed to by **`MyHT`**.

```
MyHT->NumBuckets = 5;
```

Part C (6 points) Assuming a 32-bit system, what are the following values?

`sizeof(HashTable)` = **8 bytes** `sizeof(Entry)` = **12 bytes**

Part D (12 points) Complete the C subroutine called `CalculateSize` shown below. It should take a pointer to a `HashTable` named **`HT`** and return the total number of Entries in the given `HashTable`. (Assume that the referenced `HashTable`'s `Buckets` have been allocated and initialized, and several `Entry` structures have already been inserted into it.) Be sure to declare and initialize any additional local variables you may need.

```
int CalculateSize(HashTable *HT) {
  int Size = 0;
  int I;
  Entry *ThisEntry;
  for (I=0; I < HT->NumBuckets; I++){
    ThisEntry = HT->Buckets[I];
    while(ThisEntry != Null){
      Size++;
      ThisEntry = ThisEntry->Next;
    }
  }
  return(Size);
}
```

Part E (10 points) Write the MIPS code implementation of the dynamically allocated array access below in the smallest number of instructions. A pointer to the array (declared below) is stored in `$3`. Variables **`W`**, **`X`**, **`Y`**, **`Z`**, and **`R`** reside in `$4`, `$5`, `$6`, `$7`, and `$2` respectively. Modify only registers `$1` and `$2`.

```
int     Array[8][5][4][16];      /* array declaration */
R = Array[Z][Y][X][W];           /* implement this */
```

Label	Instruction	Comment
	addi $1, $0, 320	# L_Y*L_X*L_W = 320
	mult $1, $7	# L_Y*L_X*L_W * Z
	mflo $1	# $1:320*Z
	sll $2, $6, 6	# L_X*L_W*Y = 64*Y
	add $1, $1, $2	# 320*Z + 64*Y
	sll $2, $5, 4	# L_W*X = 16*X
	add $1, $1, $4	# 320*Z + 64*Y + 16*X
	add $1, $1, $4	# 320*Z+64*Y+16*X+W
	sll $1, $1, 2	# scale by 4
	add $1, $1, $3	# add array base
	lw $2, 0($1)	# R = array element loaded

Problem 4 (3 parts, 25 points) **Garbage Collection**

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	$6024	6164	0
6008	$6132	6040	$6120	6072	$6004	6104	16	6136	9	6168	0
6012	16	6044	$6080	6076	8	6108	5	6140	$6072	6172	0
6016	80	6048	16	6080	$6024	6112	$6148	6144	20	6176	0
6020	8	6052	$6092	6084	$6052	6116	8	6148	6046	6180	0
6024	25	6056	$6024	6088	4	6120	32	6152	8	6184	0
6028	$6004	6060	0	6092	$6080	6124	$6024	6156	26	6188	0

Part A (10 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, 6080, 6024, 6004, 6132, 6072

Part B (5 points) If a reference counting garbage collection strategy is being used, what would be the reference count of the object at address $6004?

Reference count of object at $6004 = 3

Part C (10 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:	none
Mark and Sweep:	6052, 6092, 6080
Old-New Space (copying):	6052, 6092, 6080

Problem 5 (2 parts, 30 points) Activation Frames

Consider the following C code fragment:

```
typedef struct {
   int x;
   int y;
} pair;

int Bar() {
   int         M = 3;
   int         N = 4;
   pair        P;
   int         Q[] = {10, 20, 30};
   int         Foo(int, pair *, int *);
   P.x = 5;
   P.y = 6;
   N = Foo(M, &P, Q);
   return(N);
}

int Foo(int A, pair *B, int *C) {
   int         D = 25;
   int         E;

   C[2] = C[1] + A;
   E = B->y + D;
   return(E);
}
```

Part A (15 points) Describe the current state of the stack <u>just before `Foo` returns to `Bar`</u>. Fill in the unshaded boxes to show `Bar's` and `Foo`'s activation frames. Include a symbolic description and the actual value (in decimal). 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 Bar's caller**	
9896	**FP of Bar's caller**	
SP, Bar's FP 9892	RV	
9888	M	3
9884	N	4
9880	P.y	6
9876	P.x	5
9872	Q[2]	~~30~~ 23
9868	Q[1]	20
9864	Q[0]	10
9860	RA	
9856	FP	9892
9852	A	3
9848	B	9876
9844	C	9864
9840	RV	31
9836	D	25
9832	E	31

Part B (15 points) Write MIPS code fragments to implement the subroutine `Foo` 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 `Foo`'s activation frame).

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

label	instruction	Comment
	`add $30, $29, $0`	`# set FP to RV slot`
	`addi $29, $29, -8`	`# make room for 2 locals`
	`addi $1, $0, 25`	`# load constant 25`
	`sw $1, -4($30)`	`# store 25 in local D`

`C[2] = C[1] + A;`

label	instruction	Comment
	`lw $1, 4($30)`	`# load C base`
	`lw $2, 4($1)`	`# load C[1]`
	`lw $3, 12($30)`	`# load A`
	`add $2, $2, $3`	`# C[1] + A`
	`sw $2, 8($1)`	`# C[2] = C[1] + A`

`E = B->y + D;`

label	instruction	Comment
	`lw $1, 8($30)`	`# load B`
	`lw $2, 4($1)`	`# load B->y`
	`lw $3, -4($30)`	`# load D`
	`add $2, $2, $3`	`# B->y + D`
	`sw $2, -8($30)`	`# E = B->y + D`

`return(E); (compute return value, deallocate locals, and return)`

label	instruction	Comment
	`addi $29, $29, 8` `(or add $29, $30, $0)`	`# deallocate locals`
	`sw $2, 0($30)`	`# store E in RV slot`
	`jr $31`	`# return to caller`