Full Name:______________________________________

CS 15-740, Fall 1997

Midterm Exam

October 20, 1997

Instructions:

- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 69 points (!).
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is OPEN BOOK. You may use any books or notes you like.

Do not write below this line

1. ________	6. ________
2. ________	7. ________
3. ________	8. ________
4. ________	9. ________
5. ________	
total ________	

Useful Information

MIPS Notation Conventions

In all of the MIPS code shown in this exam, we will follow the notation conventions used in the lecture notes, especially regarding the distinction between delayed and normal branches. That is:

- Any jump or branch instruction written with UPPER CASE letters is a *delayed* branch. The instruction after it in the program will be executed regardless of whether or not the branch is taken.
- Any jump or branch instruction written with lower case letters is a *conventional* branch. The instruction after it in the program will not be executed if the branch is taken.
- All assembler directives, e.g., `.frame` are omitted.

Data Sizes and Alignments

You may assume the following sizes (in bytes) for different basic arithmetic data types:

C Type	Bytes
`short`	2
`int`	4
Pointer	4
`char`	1
`float`	4
`double`	8

All data of basic arithmetic data types must be aligned.

The next problem concerns the following scenario:

You are charged with modifying part of a large C program, and you come across the following code:

```
typedef struct {
  int left;
  a_struct a[CNT];
  int right;
} b_struct;


void test(unsigned i, b_struct *bp)
{
  int n = bp->left + bp->right;
  a_struct *ap = &(bp->a[i]);
  ap->x[3] = n;
}
```

Unfortunately, the '`.h`' file defining the compile-time constant `CNT` and the structure `a_struct` are in files for which you do not have access privileges. Fortunately, you have access to a '`.o`' version of your code. You are able to disassemble this with the `dis` program. Here is the disassembly of the procedure `test`, modified to use our branch notation:

```
    lw      r3,0(r5)
    lw      r2,224(r5)
    nop
    addu    r3,r3,r2
    sll     r2,r4,1
    addu    r2,r2,r4
    sll     r2,r2,3
    addiu   r2,r2,8
    addu    r5,r5,r2
    JR      r31
    sw      r3,20(r5)
```

Problem 1. (8 points):

Using your reverse engineering skills, deduce the following:

A. The size of the structure `a_struct`.

B. The alignment requirements for `a_struct`.

C. The value of `CNT`.

D. A diagram of the complete layout for `a_struct`. [You may not know the names of all of the fields for this structure, but you can determine their types.] Mark off the different fields and indicate as much information as you can. Clearly indicate the end of the structure by crossing off any unused bytes. The diagram allows a maximum of 36 bytes.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The next problem is based on the following C declaration:

```
struct BddStruct {
    struct BddStruct* next;     /* for chained hashing */
    struct BddStruct* left;     /* left child of this node */
    struct BddStruct* right;    /* right child of this node */
    unsigned short    var;
    unsigned short    flags;
};

typedef struct BddStruct* BddNode_t;

/* a BDD hash table of N_BDD_BINS bins */
BddNode_t hashTable[N_BDD_BINS];
```

The initial register to variable mapping for each of the assembly code section is

```
r2 = hashTable
r4 = i
```

The assembly instruction

```
lhu rt, offset(base)
```

loads an unsigned halfword from memory address (base + offset) into register rt.

Problem 2. (8 points):

This problem consists of eight C references followed by six MIPS assembly code sections (labeled A to F). Next to each of the C references, please indicate which assembly code section places the value of that C reference into register r3. If no match is found, please write "NONE" next to the C reference. (Note: it is also possible that an assembly code section has no matching C reference. In this case, you can just ignore that assembly code section.)

WARNING: Answers left blank will be marked as incorrect.

C References:

```
_______ hashTable[0]->var                     _______ hashTable[2]->var

_______ hashTable[0]->flags                   _______ hashTable[i]->next->left

_______ hashTable[i]->next->left->next        _______ hashTable[i]->right->left

_______ hashTable[i]->next->next->next        _______ hashTable[i]->left->right
```

Assembly Code Sections:

```
Assembly Code A:
      sll       r4,r4,2
      addu      r2,r2,r4
      lw        r2,0(r2)
      nop
      lhu       r3,12(r2)
```

```
Assembly Code B:
      lw        r2,0(r2)
      nop
      lhu       r3,12(r2)
```

```
Assembly Code C:
      sll       r4,r4,2
      addu      r2,r2,r4
      lw        r2,0(r2)
      nop
      lw        r2,0(r2)
      nop
      lw        r2,0(r2)
      nop
      lw        r3,0(r2)
```

```
Assembly Code D:
      sll       r4,r4,2
      addu      r2,r2,r4
      lw        r2,0(r2)
      nop
      lw        r2,8(r2)
      nop
      lw        r3,4(r2)
```

```
Assembly Code E:
      sll       r4,r4,2
      addu      r2,r2,r4
      lw        r2,0(r2)
      nop
      lw        r2,0(r2)
      nop
      lhu       r3,14(r2)
```

```
Assembly Code F:
      sll       r4,r4,2
      addu      r2,r2,r4
      lw        r2,0(r2)
      nop
      lw        r2,4(r2)
      nop
      lw        r3,8(r2)
```

The following problem concerns the relation between CISC and RISC instruction sets. Consider the following, not very useful function written in C:

```
void test_fact(void)
{
  int f[10];
  int n;
  for (n = 0; n < 10; n++) {
    f[n] = rfact(n);
  }
}
```

The following shows a realization of this function for a VAX architecture:

```
_test_fact:
        .word 0x40
        subl2 $40,sp          (1)
        clrl r6                (2)
L9:
        pushl r6               (3)
        calls $1,_rfact        (4)
        movl r0,-40(fp)[r6]    (5)
        incl r6                (6)
        cmpl r6,$9             (7)
        jleq L9                (8)
        ret                    (9)
```

The numbers in parentheses are for reference purpose only. They would not be in the actual assembly code. A call to this function would take the form:

```
        calls $0, _test_fact  (10)
```

Here's a little table of relevant VAX operations

Form	Name	Effect
`subl2` *op1*, *op2*	Subtract longword	*op2* <-- &*op1* − *op2*
`clrl` *op1*	Clear longword	*op1* <-- 0
`pushl` *op1*	push longword	push *op1* onto stack
`calls` *cnt*, *proc*	procedure call	call *proc* with *cnt* arguments
`movl` *op1*, *op2*	move longword	*op2* <-- &*op1*
`incl` *op1*	Increment longword	*op1*++
`cmpl` *op1*, *op2*	compare longword	compare *op1* and *op2*
`jleq` *dest*	jump if less than or equal to	jump to *dest* if comparison <=
`ret`	return	procedure return

Problem 3. (10 points):

The following shows the code for a MIPS implementation of this function in our stylized assembly code format. To the left of each instruction line is a space where you are to indicate which line of the VAX code has a comparable effect in the realization of the overall function.

You should indicate exactly one choice for each line of MIPS code. If you feel there are multiple possible choices, or no clear choice, choose the line that most nearly matches. Remember that the call to `test_funct` (line 10) is one of your possible choices.

```
test_fact:
___    addiu     $sp,$sp,-72
___    sw        $17,60($sp)
___    sw        $16,56($sp)
___    move      $16,$0
___    addiu     $17,sp,16
___    sw        $31,64($sp)
loop:
___    JAL       rfact
___    move      $4,$16
___    sll       $3,$16,2
___    addu      $3,$3,$17
___    sw        $2,0($3)
___    addiu     $16,$16,1
___    slti      $2,$16,10
___    BNE       $2,$0,loop
___    nop
___    lw        $31,64(sp)
___    lw        $17,60(sp)
___    lw        $16,56(sp)
___    JR        $31
___    addiu     $sp,$sp,72
```

Problem 4. (10 points):

Assume that the architecture of this problem is the pMIPS architecture from Assignment 2, with the following properties:

- The processor handles hazards by forwarding results whenever possible.
- There is no delay slot after loads. The instruction following a load may use the result of that load. The processor stalls to allow this only if necessary.

You are to draw the pipeline chart for the following fragment of hazardous code.

```
lw    $8,64($3)
addu  $5,$3,$0
addu  $2,$8,$5
lw    $5,0($2)
addu  $3,$5,$0
lw    $7,0($3)
sw    $7,4($3)
```

Please fill in the appropriate boxes with the letters F, D, E, M or W. Where forwarding happens, draw an arrow from the stage supplying the result to the stage using the result. Where a stage gets stalled, just fill in the same letter twice. To get you started, we have already provided the first few timesteps:

Instr	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}
lw $8,64($3)	F	D	E	M								
addu $5,$3,$0		F	D	E								
addu $2,$8,$5			F	D								
lw $5,0($2)				F								
addu $3,$5,$0												
lw $7,0($3)												
sw $7,4($3)												

Problem 5. (8 points):

Consider the following MIPS assembly language loop that adds a constant to each element of an array of floats:

```
# on entry: r4 = &a, r2 = n, f12 = c
R:  lwc1   f0,0(r4)    # f0 = a[i]
A:  add.s  f0,f0,f12   # f0 += c
S:  swc1   f0,0(r4)    # a[i] += c
I:  addiu  r4,r4,4     # i++
B:  BNE    r4,r2,R     # loop while i < n
    nop
```

Recall that a MIPS floating point add requires 2 cycles, and assume that the pipeline stalls if there is a dependent ALU operation in the load delay slot of the lwc1 instruction. Also, assume that this code is being executed by a real MIPS machine, rather than our simulator, so there is no stalling or cancelling involved in handling a branch instruction.

A. How many clock cycles does one iteration of the above loop require? (Including the nop in the branch delay slot).

B. Suppose that you are only allowed to reorder instruction within one basic block (i.e., no loop unrolling or software pipelining), and that you are allowed to change the displacement field in load and store operations. Then what is the optimal schedule for this loop? Indicate the optimal schedule on the lines below, using the instruction label to identify each instructions.

1. ______ 2. ______ 3. ______ 4. ______ 5. ______

C. How many clock cycles does one iteration of your optimized loop in part B require.

The following 2 problems concern the C function `average()`. This function includes a for loop which has three sections: LOAD, DIVIDE, and ACCUMULATE.

```
typedef struct class{
  int nstudents;
  float *scores;
  float max_score;
  float average;
} class;

void average(class *my_class)
{
    int i;
    float max = my_class->max_score;
    float temp = 0.0;
    float *scores = my_class->scores;

    for (i = 0; i < my_class->nstudents; i++)
    {
      /* LOAD */
      float sload = *(scores++);

      /* DIVIDE */
      float grade = sload / max;

      /* ACCUMULATE */
      temp += grade;
    }
    my_class->average = temp / my_class->nstudents;
}
```

Problem 6. (5 points):
The following is the MIPS assembly code for a version of `average()` which uses three-stage software pipelining. It assumes that there are at least two students in the class. This code has been modified to use our conventions for describing branch instruction types.
Identify the loop by drawing a bracket in the left margin to clearly show which instructions make up the loop. On the line next to each instruction in the loop, indicate the section of code that the instruction belongs to by writing one of the abbreviations from the table below. You should label every instruction in the loop, and leave all other instructions unlabelled.

T	for-loop test
U	for-loop update
L	LOAD stage
D	DIVIDE stage
A	ACCUMULATE stage

```
____  0x5c: li r5,2
____  0x60: lwc1 $f6,8(r4)
____  0x64: mtc1 r0,$f2
____  0x68: lw r3,4(r4)
____  0x6c: nop
____  0x70: lwc1 $f0,0(r3)
____  0x74: nop
____  0x78: div.s $f4,$f0,$f6
____  0x7c: addiu r3,r3,4
____  0x80: lwc1 $f0,0(r3)
____  0x84: lw r2,0(r4)
____  0x88: nop
____  0x8c: move r6,r2
____  0x90: slt r2,r5,r2
____  0x94: BEQ r2,r0,0xb8
____  0x98: addiu r3,r3,4
____  0x9c: add.s $f2,$f2,$f4
____  0xa0: div.s $f4,$f0,$f6
____  0xa4: lwc1 $f0,0(r3)
____  0xa8: addiu r5,r5,1
____  0xac: slt r2,r5,r6
____  0xb0: BNE r2,r0,0x9c
____  0xb4: addiu r3,r3,4
____  0xb8: add.s $f2,$f2,$f4
____  0xbc: div.s $f0,$f0,$f6
____  0xc0: add.s $f2,$f2,$f0
____  0xc4: lwc1 $f0,0(r4)
____  0xc8: nop
____  0xcc: cvt.s.w $f0,$f0
____  0xd0: div.s $f0,$f2,$f0
____  0xd4: JR r31
____  0xd8: swc1 $f0,12(r4)
```

Problem 7. (5 points):

This problem concerns the software pipelined loop in the previous problem. When you are asked to identify loop stages, do so by giving the name of the stage and the iteration of the initial unpipelined loop that it belongs to, e.g. LOAD 1 or ACCUMULATE i+1. Loop iterations are numbered starting with zero. Use n to denote the value of `my_class->nstudents`.

A. Which loop stages make up the prologue?

B. Which loop stages make up the epilogue?

C. Consider the LOAD stage that is executed during the ith iteration of the software-pipelined loop. In which iteration of the original loop would this LOAD stage have been executed?

D. Answer the previous question for the DIVIDE stage of the loop.

E. Answer the previous question for the ACCUMULATE stage of the loop.

Problem 8. (7 points):

In this problem, all numbers are 32-bit. Signed numbers are represented in 2's complement format. In C, constants are consider to be signed, unless suffixed with 'u' or 'U.' All arithmetic operations (including comparisons) can be performed over either signed or unsigned arguments. When an operation has arguments of different types (e.g., signed and unsigned), the unsigned values will be "promoted" to signed.
The variables used in this problem are declared and initialized as follows:

```
int x = foo();
int y = bar();
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
```

For each of the C statements below, circle T if it is always true, or F if it can be false. If you circle F, give a counterexample *using only values from the table below*. Refer to the table entries by their row and column, e.g. write B2 for the value 80000001. Be sure to indicate the value of each variable in the statement. If there is more than one counterexample for a statement, choose one. You may use the same counterexample for more than one statement.
Numbers in the table are given in hexadecimal.

	1	2	3
A	12345678	00000001	7FFFFFFF
B	FFFFFFFF	80000001	00000000
C	80000000	10000000	F0000000

A. T F `uy - y = 0`

B. T F `ux >= 0`

C. T F `(x < 0) --> ((x * 2) != 0)`

D. T F `(x > 0) --> ((x << 30) > 0)`

E. T F `(y < 17) --> (-y > -17)`

F. T F `(x > 0) --> ((x + 2) > 2)`

G. T F `ux <= -1`

Problem 9. (8 points):

Suppose we want to devise a floating point format that fits in only 4 bits (!). Here's the plan:

- There is no sign bit. Only nonnegative values can be represented.
- The two-bit exponent field uses a biased representation, with a bias of 2. When the exponent is 00, the mantissa is not normalized.
- Unlike the IEEE standard, we will not represent special values infinity and NaN. Exponent pattern 11 is therefore available for encoding numeric values.
- The two-bit significand field has an implied leading 1. for the normalized case.

A. For the exponent, each two-bit pattern encodes an exponent value 2^E. Fill in the following table showing the possible numeric values for each bit pattern. Express these as exact rational numbers, e.g., 7, 7/16, or 7/4, rather than their decimal equivalents. Note that you are to express the total weight of the exponent field 2^E rather than the exponent E.

Bit Pattern	Exp. Weight 2^E
00	
01	
10	
11	

B. For the mantissa, each two-bit pattern encodes a mantissa value m, where the value will depend on whether or not the number is normalized. Fill in the following table showing the two possible numeric values for each bit pattern. Express these as exact rational numbers, e.g., 7, 7/16, or 7/4, rather than their decimal equivalents.

Bit Pattern	Normalized	Denormalized
00		
01		
10		
11		

C. Fill in the following table indicating the numeric value represented by each possible 4-bit pattern. Express the entries as exact rational numbers.

		Significand			
		00	01	10	11
Exponent	00				
	01				
	10				
	11				