Full Name:____________________________________

Andrew ID:____________________________________

CS 15-347, Spring 1997

Exam 2

March 20, 1997

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew ID *on every page indicated*.
- 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 78 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. _________	2. _________
3. _________	4. _________
5. _________	6. _________
7. _________	8. _________
_________	_________
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.

Full Name:____

Andrew ID:____

Problem 1. (4 points):

Sometimes a bit is just a bit.

Consider representing numbers with a 3-bit word size. We will consider four different encodings:

Unsigned Only nonnegative numbers are encoded.

Two's Complement Signed integers are encoded in two's complement form.

Biased Signed integers are encoded with a bias of 4.

Sign Magnitude The most significant bit indicates the sign, while the remaining bits indicate the absolute value.

Fill in the following table giving the numeric value for all eight possible bit patterns for each of the four encodings.

Bits	Unsigned	Two's Comp.	Biased	Sign Magnitude
000				
001				
010				
011				
100				
101				
110				
111				

Full Name:______________________________________
Andrew ID:______________________________________

Problem 2. (14 points):

Pipeline mania!

Assume that the architecture of this problem is the pMIPS architecture from Lab 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    $7,0($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 where the result is generated to the stage where the result is used. 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 $7,0($3)												

Full Name:_______________________________________

Andrew ID:_______________________________________

Problem 3. (15 points):

To bubble or not to bubble? That is the question.

Consider the pipelined pMIPS architecture used in Lab 2. Remember that it has the following properties:

- Forwarding is used to handle hazards wherever possible. When hazards cannot be resolved through forwarding, the appropriate pipe registers are stalled.
- When a branch is taken, the PC pipe register is written during the MEM stage of the instruction, and the new PC is used on the next cycle. A branch has one delay slot; the instructions after the delay slot that have been fetched are canceled.

Suppose that we add the instructions BEQL and BNEL to the pMIPS architecture the same way as BEQ and BNE except that we cancel the delay slot when the branch is not taken. (Recall from Lab 1, BEQL behaves just like BEQ except that when the branch is not taken, the instruction in the delay slot is canceled. Similarly, BNEL behaves just like BNE except that when the branch is not taken, the instruction in the delay slot is canceled.) Consider the following disassembled MIPS instructions. Assume that we start executing this instruction sequence at clock cycle 1.

```
        addui     r5, r0, 1       # r5 = 1
        addui     r6, r0, 1       # r6 = 1

$loop:  andi      r7, r5, 0xff    # r7 =  r5 % 16
        BNEL      r7, r0, $cont   # if ((r5 % 16) != 0) goto $cont (hint: taken)
        addui     r5, r5, 1       # r5++
$cont:  BEQL      r5, r6, $loop   # if (r5 == r6) goto $loop (hint: not taken)
        addui     r6, r6, 1       # r6++

        sll       r5, r6, 2       # r5 = r6 * 4
        or        r3, r2, r5      # r3 = r2 | r5
        xor       r1, r2, r3      # r1 = r2 xor r3
        sub       r4, r5, r6      # r4 = r5 - r6
```

Full Name:______________________________________

Andrew ID:______________________________________

The following table contains a snapshot of the pipeline stages for each instruction executed from clock cycle 3 to 13. Please fill in ALL the blanks in the table with the following key to label the pipeline stages:

- **F, D, E, M,** or **W**: The instruction is in the IF, ID, EXE, MEM, or WB stage and neither the stall signal nor the bubble signal is asserted.
- **BF, BD, BE, BM,** or **BW**: The instruction is cancelled in the specified stage by the bubble signal.
- **—**: None of the above. This will include the cases where the instruction has not been fetched yet, the instruction has exited the pipeline, or the original instruction has been canceled earlier.

Note: Any blank left unfilled will be marked as an INCORRECT answer. And please do not forget to fill in the instructions for the blanks in the first column.

Instruction	**Clock cycle**										
	3	4	5	6	7	8	9	10	11	12	13
andi ...	F	D	E	M	W	—	—	—	—	—	—
BNEL ...	—	F	D	E	M	W	—	—	—	—	—
addui r5...	—	—	F	D	E			—	—	—	—
BEQL ...	—	—	—	F	D					—	—
addui r6...	—	—	—	—	F						—
	—	—	—	—	—	F					
	—	—	—	—	—	—	F				
	—	—	—	—	—	—	—	F			
	—	—	—	—	—	—	—	—	F		

Full Name:____

Andrew ID:____

Problem 4. (6 points):

No man is an island.

Consider the following MIPS assembly language code fragment:

```
# on entry: r4 = &a, f12 = c
S1:  lwc1   f0,0(r4)   # f0 = a[i]
S2:  add.s  f2,f0,f12
S3:  lwc1   f0,4(r4)   # f0 = a[i+1]
S4:  add.s  f2,f0,f12
```

There is at most dependence between any pair of statements in this code fragment. In the table below, indicate the type of dependence, using the following key:

d:	data dependence (true or flow dependence)	a:	antidependence
o:	output dependence	n:	none of the above

For example, if there is a data dependence between S1 and S2, then enter a "d" in row S1, column S2. If there is an antidependence between S1 and S2, then enter an "a" in that position, and so on. Fill all blanks; any blank left unfilled will be marked as an INCORRECT answer.

Hint: Recall that a data dependence involves the flow of a specific *value* from a producer to a consumer.

	S1	S2	S3	S4
S1	—-			
S2	—	—		
S3	—	—	—	
S4	—	—	—	—

Full Name:____

Andrew ID:____

Problem 5. (5 points):

Unemployed compiler writers' motto: "We do our code the old fashioned way, one loop at a time."

Recall from lecture that a loop is parallel iff it contains no loop-carried dependences. Indicate whether each of the following loops is a parallel loop by circling either "parallel" or "not parallel".

```
A. parallel  not parallel    for (i = 1; i <= 5; i++)
                                y[i] = y[i];


B. parallel  not parallel    for (i = 1; i <= 5; i++)
                                y[i] = y[i-1];


C. parallel  not parallel    for (i = 1; i <= 5; i++)
                                y[i] = y[2*i - 10];


D. parallel  not parallel    for (i = 1; i <= 5; i++)
                                y[i] = y[2*i - 4];


E. parallel  not parallel    for (i = 1; i <= 5; i++)
                                y[i] = y[10-i];
```

Full Name:_______________________________________

Andrew ID:_______________________________________

Problem 6. (8 points):

Sometimes a busy schedule is a good thing!

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]
G:  add.s  f0,f0,f12  # f0 += c
T:  swc1   f0,0(r4)   # a[i] += c
I:  addiu  r4,r4,4    # i++
H:  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.

Full Name:_____________________________________

Andrew ID:_____________________________________

Problem 7. (16 points):

Q: What's purple and commutes? A: An abelian grape.

In this problem, all numbers are 32-bit. Signed numbers are represented in 2's complement format. Recall from lecture that the promotion rules tell us signed integers will be promoted to unsigned when necessary. 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 expressions below, circle T if it is always true, or F if it can be false. If you circle F, give a concise counterexample.

A. `T F (x < 0) --> ((x * 2) < 0)`

B. `T F ux >= 0`

C. `T F (x & 7 == 7) --> (x << 30) < 0`

D. `T F (x < 0) --> ((x << 1) != 0)`

E. `T F ux > -1`

F. `T F (x > y) --> (-x < -y)`

G. `T F (x * x) >= 0`

H. `T F (x > 0) && (y > 0) --> (x + y) >= 0`

Full Name:_______________________________________
Andrew ID:_______________________________________

Problem 8. (10 points):

Q: How can a point float?

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. Exponent 00 is used to represent denormalized values.
- 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. Each possible two-bit pattern therefore encodes either an exponent weight 2^E or a mantissa value m, where the mantissa value will depend on whether or not the number is normalized. Fill in the following table showing the three possible numeric values. 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	Mantissa m	
		Normalized	Denormalized
00			
01			
10			
11			

B. 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
	00				
Exponent	01				
	10				
	11				