Your name ______________________________ login cs61c-________

Question 1, Deja Vu all over again (5 points):

1 point: Using two's complement, saturating arithmetic, add the following 8 bit numbers together.

+ 00110101
+ 00101111
+ 01011011

1 point: Which IEEE rounding mode would you like us to use for dealing with fractional points, in order to maximize your score: round to $+\infty$, round to $-\infty$, truncate, or round to even?

+\infty, aka round up.

1 point: Why does this not work as a translation of \texttt{lui} \texttt{imm}? \texttt{imm} is a 32 bit quantity, \texttt{high} is the upper 16 bits of \texttt{imm}, \texttt{low} is the lower 16 bits of \texttt{imm}:

\begin{verbatim}
lui $rd \text{ } \text{imm}  
addiu $rd $d \text{ } \text{low}  # Addiu sign extends the immediate
\end{verbatim}

1 point: What is the value of this 32 bit, two's complement number?

\begin{verbatim}
11111111 11111111 11111111 11111111 11111111 11111111
\end{verbatim}

-13

1 point: What registers must be restored to their preexisting values when a function returns, according to the MIPS calling convention?

$gp$, $lo$, $hi$, $gp$. Saying just $gp$ and the saved registers is sufficient. You can argue for $fn$ as well, so that was accepted.

READ AND SIGN THIS:

I certify that my answers to this exam are all my own work, and that I have not discussed the exam questions or answers with anyone prior to taking this exam.

If I am taking this exam late, I certify that I have not discussed the exam questions or answers with anyone who has knowledge of the exam.

I also certify that I was not kidnapped by evil two headed alien Elvis clones for use in their diabolical experiments.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>total</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1
Question 2 (6 points):

Short questions on performance and I/O.

1 point: How many interrupts will be required to read 100 bytes of input, one byte at a time, using polling based I/O?
0. Polling does not USE interrupts

1 point: How many interrupts will be required to read 100 bytes of input, one byte at a time, using interrupt driven I/O?

1 point: How many interrupts will be required to read 100 bytes of input, one byte at a time, using DMA transfer?

1 point: Gill Bate’s new operating system, Macroshaft WinBloxs 00 requires 10 minutes to start up. 40% of this time is used to detect and remove non Macroshaft programs. If the Department of Justice forces the removal of this portion of the operating system, what is the best time the modified operating system (the version without the portion which detects non Macroshaft programs) requires to boot?

It no longer boots without this critical code. Or it takes 6 minutes

1 point: Inbell computer corporation claims that their new Multipersonality Extension (MPX instructions) based CPUs perform some operations, those represented by MPX instructions, up to 10 times faster, but all non MPX instructions are unaffected. Older programs do not use MPX instructions. What is the maximum percentage improvement which a MPX processor offers when running old programs?

0% improvement. The old code does not use MPX instructions

1 point: A RISC machine has an average CPI of 1.5 and a clock rate of 500 MHz. How long will it take to execute a 100,000,000 instruction program?

3 seconds, or 300 ms

Question 3 (6 points):

Short questions on boolean logic, circuits, computer components, etc.

1 point: Which of the following boolean expressions are true?

A or 0 = 0, A or 0 = 1, A and ~A = 0, A and ~A = 1

A and ~A = 0 is correct.

2 points: Give the truth table for the following boolean function and draw a circuit which implements it.

O = AB or AC or B C

A B C | O
-------
0 0 0 | 0
0 1 0 | 0
0 1 1 | 1
1 0 0 | 0
1 0 1 | 1
1 1 0 | 1
1 1 1 | 1

2 points: A state machine has 3 states, A, B, and C, and one input I. If the machine is in state A and I is true, it transitions to state B, otherwise it stays in state A. If the machine is in state B and I is true, it transitions to state C, otherwise it transitions to state A. If the machine is in state C it will always stay in state C. Draw the state transition diagram for this state machine

1 point: Which of the following tasks does the ALU perform: Store register value, perform arithmetic operations, perform comparisons, load data from memory?

Arithmetic operations and comparisons.
Question 4 (6 points):

Consider this C program definition:

```c
struct dispatchElement{
    char *name;
    int value;
    int (*fn)(int);
};
int opcode(int i);
int dispatch(struct dispatchElement *table, int operation){
    return (table[opcode(operation)].fn)(operation);
}
```

Translate the function `dispatch` into MIPS assembly, using the register calling convention. Hint: draw the layout for a `struct dispatchElement` on the back side to make sure you fetch the correct fields of the structure.

```
dispatch:
    addi $sp $sp -12 # prolog
    sw $ra 0($sp)
    sw $s0 4($sp)
    sw $s1 8($sp)
    move $s0 $a0 # save table
    move $s1 $a1 # and operation
    move $a0 $a1 # Call
    jal operation # operation
    li $t0 12 # Each dispatchElement takes 12 words, so multiply by 12
    add $t0 $t0 $s0 # Add offset to table
    lw $t1 8($t0) # fn is the 3rd field
    move $a0 $s1 # Get op
    jalr $t0 # call fn
    lw $ra 0($sp) # and return
    lw $s0 4($sp)
    lw $s1 8($sp)
    addi $sp $sp -12
    jr $ra
```

Question 5 (6 points):

Ben Bitdiddle has written the following faulty interrupt handler. The device in question can have interrupts enabled by writing a 1 to the address 0xffff0010, and disabled by writing a 0 to that address. When this device generates an interrupt, the value 0xf00d is placed in the cause register (coprocessor0 register $13). No other interrupt will generate this value for the cause register. He wants to allow other interrupts to proceed while his interrupt handler is processing this interrupt, but does not want to receive interrupts from the device in question. Only his interrupt handler manipulates the device’s interrupt enable register, so he doesn’t have to worry about someone else reenabling the device’s interrupt. Unfortunately there are 3 bugs in his code. Identify each bug, correct it in the code, and explain why each bug is a problem. Code written out as ./... is correct. He is allowed to use the stack for storage. His bugs do not involve his enabling and disabling of interrupts or the manipulation of registers to enable or disable interrupts, but involve not correctly saving a register, enabling or disabling interrupts at the wrong time or in the wrong order, and using registers at inappropriate times. All comments correctly describe the behavior of the commented code.

```
.ktext 0xb8000000 # Forces interrupt routine below to # be located at the right spot

intrp:
1) addi $sp $sp -20 # Get some stack space
2) mfc0 $k0 $13 # See what caused the interrupt
3) li $t1 0xf00d # If it is not ours, skip on
4) bne $k0 $t1 other_interrupt
5) sw $t0 0($sp) # Save $t0 and $t1
6) sw $t1 4($sp)
7) mfc0 $k0 $14 # Saving exception PC
8) sw $k0 8($sp) # Saving status register
9) mfc0 $k0 $12 # which will reenable interrupts
10) sw $k0 12($sp) # saving status register
11) ori $k0 $k0 1 # Set interrupt bit to 1 and relpace
12) mtc0 $k0 $12 # which will reenable interrupts
13) li $t0 0xffff0010 # Turn off the device
14) li $t1 0 # so it doesn’t send more
15) sw $t1 0($t0) # Interrupts
    # Continues on next page
```
The body of the code is correct

20) lw $t0 12($sp) # Load the status and
21) mtc0 $k0 $12 # Disable interrupts

22) li $t0 0xffff # Reenable the device
23) li $t1 1 # so it will send interrupts
24) sw $t1 0($t0)

25) lw $k0 8($sp) # Restore registers
26) lw $t0 0($sp) # And return to $k0
27) lw $t1 4($sp)

28) rfe
29) jr $k0

Other interrupt: # This code is OK, and will
... # return on its own

Bug #1 is around line(s) 1 and 27. The problem is:

forgetting to save and restore $at

Bug #2 is around line(s) 12-15. The problem is:

reenabling interrupts before disabling interrupts for the device

Bug #3 is around line(s) 20-21. The problem is:

using $k0 while interrupts are still enabled. $t0 should be used instead.

Bug #4 is around line(s) 27. The problem is:

Nick forgot to increment the stack pointer when he was done