Lab 3 solution
Numbers, Instructions, Logical Operations
CS 350: Computer Organization & Assembler Language Programming
Due Sat Feb 16, 11:59 pm

Problems [50 points total]
1. (Hexadecimal numbers)
   1a. (Exercise 2.8) Let's translate ABCDEF12₁₆ to decimal one digit at a time:

   A₁₆ = 10
   AB₁₆ = 10 * 16 + 11 = 171
   ABC₁₆ = 171 * 16 + 12 = 2748
   ABCD₁₆ = 2748 * 16 + 13 = 43981
   ABCDE₁₆ = 43981 * 16 + 14 = 703710
   ABCDEF₁₆ = 703710 * 16 + 15 = 11259375
   ABCDEF1₁₆ = 11259375 * 16 + 1 = 180150001
   ABCDEF12₁₆ = 180150001 * 16 + 2 = 2882400018

   1b. 1010 0111 1100 0011 1101 1001 0110 1110₂ = A7C3D96E₁₆

2. (Translate MIPS code to C)
   ```
   addi  $s1, $s0, 4  # g = f + 4
   lw    $s2, 8($s6)  # h = A[2]
   add   $t0, $s2, $s1  # t0 = A[2] + g
   sw    $t0, 12($s7)  # B[3] = t0
   ```

3. Exercise 2.9 (Translate C code to MIPS) Let's translate B[g] = A[i] + A[j] in steps. (You didn't have to do this; I'm doing it hoping you'll find it useful.) [2/23: Misread B[8] as B[g].]

   Step 1 (very close to C code)
   ```
   t0 = A[j]
   t1 = A[i]
   t1 = A[i] + A[j]
   B[g] = t1  # B[8] = t1 [2/23]
   ```

   Step 2 (translate to addresses)
   ```
   t0 = word at (address of A) + 4*j
   t1 = word at (address of A) + 4*i
   t1 = t1 + t0  # A[i] + A[j]
   word at (address of B) + 4*g = t1  # 4*8 not 4*g [2/23]
   ```
Step 3 (break up into instruction-sized pieces)

\[ t2 = 4 \cdot j \]  
# offset for index \( j \)

\[ t2 = t2 + \text{address of } A \]  
# address of \( A[j] \)

\[ t0 = \text{word at } t2 \]  
# \( t0 = A[j] \)

\[ t3 = 4 \cdot i \]  
# offset for index \( i \)

\[ t3 = t3 + \text{address of } A \]  
# address of \( A[i] \)

\[ t1 = \text{word at } t3 \]  
# \( t1 = A[i] \)

\[ t1 = t1 + t0 \]  
# \( t1 = A[i] + A[j] \)

\[ t4 = 4 \cdot g \]  
# offset for index \( g \)

\[ t4 = t4 + \text{address of } B \]  
# address of \( B[g] \)

\[ \text{word at } t4 = t1 \]  

Step 4 (convert to assembler)

# Assumptions: \( s1 = g, s3 = i, s4 = j, s6 = \text{addr } A, \)  
# \( s7 = \text{addr } B \)

\[ sll \quad t2, s4, 2 \]  
# offset for index \( j \)

\[ \text{add } t2, t2, s6 \]  
# address of \( A[j] \)

\[ lw \quad t0, 0(t2) \]  
# \( t0 = A[j] \)

\[ sll \quad t3 = s3, 2 \]  
# offset for index \( i \)

\[ \text{add } t3, t3, s6 \]  
# address of \( A[i] \)

\[ lw \quad t1, 0(t3) \]  
# \( t1 = A[i] \)

\[ \text{add } t1, t1, t0 \]  
# \( t1 = A[i] + A[j] \)

\[ sll \quad t4, s1, 2 \]  
# offset for index \( g \)

\[ \text{addi } t4, \text{zero}, 32 \]  
# offset for index \( g \) \[2/23\]

\[ \text{add } t4, t4, s7 \]  
# address of \( B[g] \)

\[ sw \quad t1, 0(t4) \]  

[2/23: Replacing the \text{sll} by the \text{addi} is the minimal fix. It’s cleaner to replace the last 3 lines by \text{sw } t1, 32(s7), however.]

4. Exercises 2.12.1–4 (Check for overflow) We’re given \( s0 = x80000000, s1 = 0xD0000000 \)

4a. If we add \( s0, s1, s1 \), the interesting hex digit is the leftmost one:

\[ 8 + D = 8_{10} + 13_{10} = 1000_{2} + 1101_{2} = 0101_{2} = 5, \text{ so } t0 = 0x50000000 \]

4b. Since \( s0 \) and \( s1 \) are both negative, their sum should be negative. Our result is positive, so our result is incorrect and did overflow occur.

4c. With \text{sub } s0, s0, s1, \text{since } -0xD0000000 = 0x2fffffff + 1 = 0x30000000, \text{we set}

\[ t0 = x80000000 - 0xD0000000 = x80000000 + 0x30000000 = x50000000 \]

4d. We have the correct result, with no overflow. In effect, we added a negative number and a positive number. This never causes overflow, since the result is closer to 0 than the starting numbers. (Overflow occurs when you try represent a number that is far from the origin.)

5. Exercise 2.14 (Numeric vs assembler representations of instructions)

To translate 0000 0010 0001 0000 1000 0000 0010 0000 0002 into an instruction, first we need the
opcode (the leftmost 6 bits): 000000 makes this an R-type instruction. Breaking up the 32 bits into instruction bit fields gives us:

```
  op  rs  rt  rd  shamt  funct
  000000 10000 10000 10000 00000 100000
```

The rs, rt, and rd fields all = 16, which indicates $s0$. The funct field is 32, which indicates add. The instruction is `add $s0, $s0, $s0`.

6. Exercises 2.19 (Bit shifting and logical operations)

Assume $t0 = 0xAAAAAAA$, $t1 = 0x12345678$.

6a. (Exercise 2.19.1) We were to use an `sll` of 8 bits instead of 44 bits.

```
sll $t2, $t0, 44  # t2 = t0 << 8 = 0  # used 8 not 44
or  $t2, $t2, $t1  # t2 = t2 | t1
```

```
1010 1010 1010 1010 1010 1010 1010 1010 = xAAAAAAA = $t0
1010 1010 1010 1010 1010 1010 1010 0000 = xAAAAAA0 = $t0 << 8
0001 0010 0100 0101 0110 1000 = 0x12345678 = $t1
```

```
1011 1010 1011 1110 1111 1111 1110 1000 0000 1111 1000 Bitwise or yields 0xBABEFE08
```

6b. (Exercise 2.19.2)

```
sll  $t2, $t0, 4
andi $t2, $t2, -1
```

```
1010 1010 1010 1010 1010 1010 1010 1010 = xAAAAAAA = $t0
1010 1010 1010 1010 1010 1010 1010 0000 = xAAAAAA0 = $t0 << 4
1111 1111 1111 1111 1111 1111 1111 1111 = -1 (sign-extended)
```

```
1010 1010 1010 1010 1010 1010 0000 1111 1000 Bitwise and yields xAAAAAAA0
```

6c. (Exercise 2.19.3)

```
srl  $t2, $t0, 3
andi $t2, $t2, 0xFFEF
```

```
1010 1010 1010 1010 1010 1010 1010 1010 = xAAAAAAA = $t0
0001 0101 0101 0101 0101 0101 0101 0101 = x15555555 = $t0 >> 3
1111 1111 1111 1111 1111 1111 1111 1111 = -1 (sign-extended)
```

```
0001 0101 0101 0101 0101 0101 0100 0101 Bitwise and yields x15555545
```