Chapter 2 Solutions: For More Practice

2.1

Instruction	Format	op	rs	rt	immediate
`addi $v0, $zero, 0`	I	8	0	2	0
`lw $v1, 0($a0)`	I	35	4	3	0
`sw $v1, 0($a1)`	I	43	5	3	0
`addi $a0, $a0, 4`	I	8	4	4	4
`addi $a1, $a1, 4`	I	8	5	5	4
`beq $v1, $zero, loop`	I	4	3	0	–5

2.5 The code loads the `sll` instruction at `shifter` into a register and masks off the shift amount, placing the least significant 5 bits from `$s2` in its place. It then writes the instruction back to memory and proceeds to execute it. The code is self-modifying; thus it is very hard to debug and likely to be forbidden in many modern operating systems. One key problem is that we would be writing into the instruction cache, and clearly this slows things down and would require a fair amount of work to implement correctly (see Chapters 6 and 7).

2.7 The new instruction treats `$s0` and `$s1` as a register pair and performs a shift, wherein the least significant bit of `$s0` becomes the most significant bit of `$s1` and both `$s0` and `$s1` are shifted right one place.

2.14 The C loop is

```
while (save [i] == k)
i += 1;
```

with `i` and `k` corresponding to registers `$s3` and `$s5` and the base of the array save in `$s6`. The assembly code given in the example is

Code before

```
Loop: sll  $t1, $s3, 2     # Temp reg $t1 = 4 * i
      add  $t1, $t1, $s6   # $t1 = address of save [i]
      lw   $t0, 0($t1)     # Temp reg $t0 = save[i]
      bne  $t0, $s5, Exit  # go to Exit if save[i] ≠ k
      addi $s3, $s3, 1     # i = i + 1
      j    Loop            # go to Loop
Exit:
```

Number of instructions executed if `save[i + m]` does not equal `k` for `m` = 10 and does equal `k` for 0 ≤ `m` ≤ 9 is 10 × 6 + 4 = 64, which corresponds to 10 complete iterations of the loop plus a final pass that goes to `Exit` at the `bne` instruction before updating `i`. Straightforward rewriting to use at most one branch or jump in the loop yields

Code after

```
      sll  $t1, $s3, 2    # Temp reg $t1 = 4 * i
      add  $t1, $t1, $s6  # $t1 = address of save[i]
      lw   $t0, 0($t1)    # Temp reg $t0 = save[i]
      bne  $t0, $s5, Exit # go to Exit if save[i] ≠ k
Loop: addi $s3, $s3, 1    # i = i + 1
      sll  $t1, $s3, 2    # Temp reg $t1 = 4 * i
      add  $t1, $t1, $s6  # $t1 = address of save[i]
      lw   $t0, 0($t1)    # Temp reg $t0 = save[i]
      beq  $t0, $s5, Loop # go to Loop if save[i] = k
Exit:
```

The number of instructions executed by this new form of the loop is 4 + 10 × 5 = 54. If `$t1` is incremented by 4 inside the loop, then further saving in the loop body is possible.

Code after further improvement

```
      sll  $t1, $s3, 2    # Temp reg $t1 = 4 * i
      add  $t1, $t1, $s6  # $t1 = address of save[i]
      lw   $t0, 0($t1)    # Temp reg $t0 = save[i]
      bne  $t0, $s5, Exit # go to Exit if save[i] ≠ k
Loop: addi $s3, $s3, 1    # i = i + 1
      add  $t1, $t1, 4    # $t1 = address of save[i]
      lw   $t0, 0($t1)    # Temp reg $t0 = save[i]
      beq  $t0, $s5, Loop # go to Loop if save[i] = k
Exit:
```

The number of instructions executed is now 4 + 10 × 4 = 44.

2.21 No solution provided.

2.22 No solution provided.

2.23 No solution provided.

2.24 Here is the C code for `itoa`, as taken from *The C Programing Language* by Kernighan and Ritchie:

```
    void reverse( char *s )
    {
        int c, i, j;

        for( i = 0, j = strlen(s)-1; i < j; i++, j-- ) {
            c=s[i];
            s[i]=s[j];
            s[j] = c;
        }
    }

    void itoa( int n, char *s )
    {
        int i, sign;

        if( ( sign = n ) < 0 )
            n = -n;
        i = 0;
        do {
            s[i++] = n % 10 + '0';
        } while( ( n /= 10 ) > 0 );
        if( sign < 0 )
            s[i++] = '-';
        s[i] = '\0';
        reverse( s );
    }
}
```

The MIPS assembly code, along with a main routine to test it, might look something like this:

```
.data
hello:.ascii  "\nEnter a number:"
newln:.asciiz "\n"
str:.space  32

    .text
```

```
reverse:  # Expects string to
          # reverse in $a0
          # s = i = $a0
          # j = $t2

    addi    $t2, $a0, -1      # j = s -1;
    lbu     $t3, 1($t2)       # while( *(j+1) )
    beqz    $t3, end_strlen
strlen_loop:
    addi    $t2, $t2, 1       #    j++;
    lbu     $t3, 1($t2)
    bnez    $t3, strlen_loop
end_strlen:                   # now j =
                              # &s[strlen(s)-1]
    bge     $a0, $t2, end_reverse # while( i < j )
          # {
reverse_loop:
    lbu     $t3, ($a0)        #    $t3 = *i;
    lbu     $t4, ($t2)        #    $t4 = *j;
    sb      $t3, ($t2)        #    *j = $t3;
    sb      $t4, ($a0)        #    *i = $t4;
    addi    $a0, $a0, 1       #    i++;
    addi    $t2, $t2, -1      #    j--;
    blt     $a0, $t2, reverse_loop # }
end_reverse:
    jr      $31

    .globl  itoa              # $a0 = n
itoa:addi   $29, $29, -4      # $a1 = s
    sw      $31, 0($29)
    move    $t0, $a0          # sign = n;
    move    $t3, $a1          # $t3 = s;
    bgez    $a0, non_neg      # if( sign < 0 )
    sub     $a0, $0, $a0      #    n = -n
non_neg:
    li      $t2, 10
```

```
itoa_loop:                      # do {
    div     $a0, $t2            #    lo = n / 10;
          #    hi = n % 10;

    mfhi    $t1
    mflo    $a0                 #    n /= 10;
    addi    $t1, $t1, 48        #    $t1 =
          #    '0' + n % 10;
    sb      $t1, 0($a1)         #    *s = $t1;
    addi    $a1, $a1, 1         #    s++;
    bnez    $a0, itoa_loop      # } while( n );
    bgez    $t0, non_neg2       # if( sign < 0 )
          # {
    li      $t1, '-'            #    *s = '-';
    sb      $t1, 0($a1)         #    s++;
    addi    $a1, $a1, 1         # }
non_neg2:
    sb      $0, 0($a1)
    move    $a0, $t3
    jal     reverse             # reverse( s );
    lw      $31, 0($29)
    addi    $29, $29, 4
    jr      $31

    .globl  main
main:addi   $29, $29, -4
    sw      $31, 0($29)
    li      $v0, 4
    la      $a0, hello
    syscall
    li      $v0, 5              # read_int
    syscall
    move    $a0, $v0            # itoa( $a0, str );
    la      $a1, str            #
    jal     itoa                #
    la      $a0, str
    li      $v0, 4
    syscall
    la      $a0, newln
    syscall
    lw      $31, 0($29)
    addi    $29, $29, 4
    jr      $31
```

One common problem that occurred was to treat the string as a series of words rather than as a series of bytes. Each character in a string is a byte. One zero byte terminates a string. Thus when people stored ASCII codes one per word and then attempted to invoice the `print_str` system call, only the first character of the number printed out.

2.25 F

2.26 None are false; all are true.

2.27 D

2.28 E

2.33 The base address of `x`, in binary, is 0000 0000 0110 0001 1010 1000 0000 0000, which implies that we must use `lui`:

```
lui   $t0, 0000 0000 0110 0001   # load the address of x into $t0
ori   $t0, $t0, 1010 1000 0000 0000
lw    $t1, 20($t0)               # $t1 = Memory[20 + $t0]
add   $t1, $t1, $t3              # $t1 = x[5] + a
sw    $t1, 16($t0)               # x[4] = x[5] + a
```

2.35

```
count = -1;
do {
   temp = *source;
   count = count + 1;
   *destination = temp;
   source = source + 1;
   destination = destination + 1;
} while (temp != 0);
```

2.36 The fragment of C code is

```
for (i=0; i<=100; i=i+1) {a[i] = b[i] + c;}
```

with `a` and `b` arrays of words at base addresses `$a0` and `$a1`, respectively. First initialize `i` to 0 with `i` kept in `$t0`:

```
add $t0, $zero, $zero   # Temp reg $t0 = 0
```

Assume that `$s0` holds the address of `c` (if `$s0` is assumed to hold the value of `c`, omit the following instruction):

```
lw $t1, 0($s0)          # Temp reg $t1 = c
```

To compute the byte address of successive array elements and to test for loop termination, the constants 4 and 401 are needed. Assume they are placed in memory when the program is loaded:

```
lw $t2, AddressConstant4($zero)    # Temp reg $t2 = 4
lw $t3, AddressConstant401($zero)  # Temp reg $t3 = 401
```

The instructions `addi` (add immediate) and `slti` (set less than immediate) can carry constants in their machine code representation, saving a load instruction and use of a register. Now the loop body accesses array elements, performs the computation, and tests for termination:

```
Loop: add $t4, $a1, $t0   # Temp reg $t4 = address of b[i]
      lw  $t5, 0($t4)     # Temp reg $t5 = b[i]
      add $t6, $t5, $t1   # Temp reg $t6 = b[i] + c
```

This `add` instruction would be `add $t6, $t5, $s0` if it were assumed that `$s0` holds the value of `c`. Continuing the loop:

```
add $t7, $a0, $t0    # Temp reg $t7 = address of a[i]
sw  $t6, 0($t7)      # a[i] = b[i] + c
add $t0, $t0, $t2    # i = i + 4
slt $t8, $t0, $t3    # $t8 = 1 if $t0 < 401, i.e., i ≤ 100
bne $t8, $zero, Loop# go to Loop if i ≤ 100
```

The number of instructions executed is $4 + 101 \times 8 = 812$. The number of data references made is $3 + 101 \times 2 = 205$.

2.39 B

2.40 All are pseudoinstructions: i. 40,000 > +32,767 =>`lui`, `ori`. ii. `beq`: Both must be registers. Exit: If $> 2^{15}$, then pseudo. iii. `sub`: Both must be registers. Even if it was `subi`, there is no `subi` in MIPS; it would generate `addi $t0,$t1, -1`;

2.41 a, b, and c. a and b could form a data reference, so they can relocate. c is a subroutine, so it can relocate. d is OK since its PC relative.

2.43 Let the program have n instructions.

Let the original clock cycle time be t.

Let N be percent loads retained.

$\text{exec}_{\text{old}} = n \times \text{CPI} \times t$
$\text{exec}_{\text{new}} = (0.76\,n + N \times 0.24\,n) \times \text{CPI} \times 1.1t$

$\text{exec}_{\text{new}} \le \text{exec}_{\text{old}}$
$(0.76\,n + N \times 0.24\,n) \times \text{CPI} \times 1.1t \le n \times \text{CPI} \times t$
$(0.76\,n + N \times 0.24\,n) \times 1.1 \le n$
$(0.76 + N \times 0.24) \times 1.1 \le 1$

$$0.76 + N \times 0.24 \le \frac{1}{1.1}$$

$$N \times 0.24 \le \frac{1}{1.1} - 0.76$$

$$N \le \frac{\frac{1}{1.1} - 0.76}{0.24}$$

$$N \le \frac{1 - 1.1 \times 0.76}{1.1 \times 0.24}$$

$N = 0.62$

We need to eliminate at least 38% of loads.

2.44 No solution provided.