3.6:  
copy: addi $v0,$zero,-1  # to fix the off-by-one error  
loop:  
  lw $v1, 0($a0)  
  sw $v1, 0($a1)  
  addi $a0,$a0,4  
  addi $a1,$a1,4  
  addi $v0,$v0,1  
  bne $v1,$zero,loop

3.13  
New gcc: (.48-.25(.33))(1)+.33(1.4)+.17(1.7)+.02(1.2) = 1.1725 CPI  
Old gcc: 1.255 CPI (from first part of problem 3.16)  

time w/o update addressing: 1.255x seconds per instruction, where x = clock frequency  
time w/ update addressing: 1.1725(1.1)x = 1.29 seconds per instruction, where x = clock frequency  

So the unmodified architecture is faster by 1.29/1.225 = 5.3%.

3.15  
a.) % of data accesses = data accesses/(instruction accesses+data accesses)  
    = .41/(1+.41) = 29.1%  
b.) All instruction accesses are reads. So,  
    %reads = (%instructions + %data(2/3))/%instructions + %data  
    = (1 + .41(2/3))/(1 + .41) = 90.3%  

3.16  
For gcc: .48(1)+.33(1.4)+.17(1.7)+.02(1.2) = 1.255 CPI  
For spice: .5(1)+.41(1.4)+.08(1.7)+.01(1.2) = 1.222 CPI

3.19  
Accumulator Version:  
load c  
add b  
store a  
add c  
store b  
load a
sub b
store d

Instructions: 8
Code bytes: $8 * 3 = 24$, since all instructions need one byte for the opcode and two for the memory address.
Data bytes: $8 * 4 = 32$, since each instruction (load, store, add, sub) requires moving a word either to or from memory.
Total memory bandwidth: 56

Memory-memory Version:
add a, b, c
add b, a, c
sub d, a, b

Instructions: 3
Code bytes: $3 * (1 + 2 + 2 + 2) = 21$, since each instruction has one byte for the opcode and two bytes for each of the memory addresses.
Data bytes: $3 * (4 + 4 + 4) = 36$, since each instruction requires moving three memory accesses: two loads and one store.
Total memory bandwidth: 57

Stack Version:
push b
push c
add
pop a
push a
push c
add
pop b
push a
push b
push d

Instructions: 12
Code bytes: $9 * (1 + 2) + 3 * 1 = 30$, since each push or pop instruction has one byte for the opcode and two bytes for each of the memory addresses, and each add or subtract instruction has just one byte for the opcode.
Data bytes: $9 * 4 = 36$, since you move 4 bytes on each push or pop.
Total memory bandwidth: 66

Load-store Version:
lw $t1, (b)$
lw $t2, (c)$
add $t0, t1, t2$
add $t1, t0, t2$
sub $t3, t0, t1$
sw $t0, (a)$
sw $t1, (b)$
sw $t3, (d)$

Instructions: 8
Code bytes: $5 * (1 + 1 + 2) + 3 * 3 = 26$, since each lw or sw instruction has one byte for the opcode, one for the register destination, and two bytes for each of the memory addresses. The add instructions will need one byte for the opcode and two bytes to specify the registers. (4 bits per register -> 12 bits -> 2 whole bytes)
Data bytes: $5 \times 4 = 20$, since each instruction requires moving three memory accesses: two loads and one store.

Total memory bandwidth: 46

Memory-memory has the smallest code size, while load-store has the lowest total memory bandwidth. These are different because specifying what you want to do in as few bytes as possible isn’t the same as minimizing memory transfers.

### 3.22

Procedure written using pseudoinstructions, assuming no delayed branches:

```assembly
atoi:    li $v0, 0  # int product = 0;
loop:    lb $t0, 0($a0)  # { char c = str[i];
     beq $t0, '
\0', _done  # if (c == '\0') done;
     blt $t0, '0', _error  # if (c invalid) error;
     subi $t0, $t0, '0'  # int digit = c - '0';
     mul $v0, $v0, 10  # product *= 10;
     add $v0, $v0, $t0  # product += digit;
     j _loop  # }
_error:  li $v0, -1  # if (error) product = -1;
_done:   jr $ra  # return product;
```

Procedure written without pseudoinstructions, assuming delayed branches and loads:

```assembly
atoi:    or $v0, $zero, $zero  # int product = 0;
ori $t2, $zero, 10  # const int ten = 10;
_loop:   lb $t0, 0($a0)  # { char c = str[I]
or $zero, $zero, $zero  # nop
beq $t0, $zero, _done  # if (c invalid) error;
slt $t1, $t0, '0'  #
beq $t1, $zero, _error  #
slt $t1, $t0, '9'+1  #
beq $t1, $zero, _error  #
subi $t0, $t0, '0'  # int digit = c - '0';
mul $v0, $t2  # product *= 10;
mflo $v0  #
j atoi_loop  #
add $v0, $v0, $t0  # product += digit; }
_error:  ori $v0, $zero, -1  # if (error) product = -1;
_done:   jr $ra  # return product;
or $zero, $zero, $zero  # nop
```

### 3.25

Procedure written using pseudoinstructions, assuming no delayed branches:

```assembly
itoa:    li $v0, 0  # int count = 0;
bge $a0, $zero, _nonnegati  # if (num < 0) {
    sb '-' , 0($a1)  # str[i] = '-';
    addi $a1, $a1, 1  # i += 1;
    addi $v0, $v0, 1  # count += 1;
    neg $a0, $a0  # num = -num; }
_nonnegati: mov $t0, $a0  # int tmpnum = num;
_loop1:   div $t0, $t0, 10  # do { tmpnum /= 10;
    addi $a1, $a1, 1  # i += 1;
    bne $t0, $zero, _loop1  # } while (tmpnum != 0);
```
```assembly
sb $zero, 1($a1) # str[i+1] = '\0';
_loop2:
  rem $t1, $a0, 10 # do { int digit = num % 10;
  div $a0, $a0, 10 # num /= 10;
  addi $t1, $t1, '0' # char c = digit + '0';
  sb $t1, 0($a1) # str[i] = c;
  addi $a1, $a1, -1 # i -= 1;
  addi $v0, $v0, 1 # count += 1;
  bne $a0, $zero, _loop2 # } while (num != 0);
  jr $ra # return count;

Procedure written without pseudoinstructions, assuming delayed branches and loads:

_cassemble:
  or $v0, $zero, $zero # int count = 0;
  orl $t2, $zero, 10 # const int ten = 10;
  slt $t3, $a0, $zero # if (num < 0) {
    beq $t3, $zero, _nonnegati #
    addi $t3, $zero, '-' # str[i] = '-';
    sb $t3, 0($a1) #
    addi $a1, $a1, 1 # i += 1;
    addi $v0, $v0, 1 # count += 1;
    sub $v0, $zero, $v0 # num = -num; }
  _nonnegati:
  add $t0, $a0, $zero # int tmpnum = num;
  _loop1:
  div $t0, $t2 # do {
    mflo $t0 # tmpnum /= 10;
    addi $a1, $a1, 1 # i += 1;
    bne $t0, $zero, _loop1 # } while (tmpnum != 0);
    or $zero, $zero, $zero # nop
    sb $zero, 1($a1) # str[i+1] = '\0';
  _loop2:
  div $a0, $t2 # do {
    mflo $a0 # int digit = num % 10;
    mfhi $t1 # num /= 10;
    addi $t1, $t1, '0' # char c = digit + '0';
    sb $t1, 0($a1) # str[i] = c;
    addi $a1, $a1, -1 # i -= 1;
    addi $v0, $v0, 1 # count += 1;
    bne $a0, $zero, _loop2 # } while (num != 0);
    or $zero, $zero, $zero # nop
    jr $ra # return count;
    or $zero, $zero, $zero # nop

Note (1) The problem does not specify whether or not to handle negative numbers. Since problem 3.22 did not require handling negative numbers, it might be okay to leave out here as well.

Note (2) The given solution works by first counting the number of digits, then printing the digits out in reverse order. Alternatively, students often tackle this problem by writing a procedure with one loop that runs by dividing the original number by 10^9, 10^8, 10^7, ... 10^1, 10^0. This direct method works, but generally requires longer and more complicated code.

3.29

sbn temp, temp, .+1 # temp = 0;
sbn temp, b, .+1 # temp = -b;
sbn a, temp, .+1 # a = a - (-b) = a + b;

3.30

sbn neg_a, neg_a, .+1 # neg_a = 0;
sbn neg_a, a, .+1 # neg_a = -a;
sbn c, c, .+1 # c = 0;
```
loop:    sbn b, one, +1    # do { b = b - 1; sbn c, neg_a, +1    #     c = c + a; sbn temp, temp, +1    #     temp = 0; sbn temp, b, loop    # } while (b > 0);

Note (1) This solution does not work if \( b = 0 \), because the problem description said to assume that \( a \) and \( b \) are greater than 0. Perfectionist students are likely to write solutions that do work for \( b = 0 \) though, so their answers would be an instruction or two longer.

A.3
No. It is never safe for a user program to use registers $k0$ or $k1$.

A.5

```
.ktext 0x80000080
sw $a0, save0
sw $a1, savel
mfc0 $k0, $13      # Move Cause into $k0
mfc0 $k1, $14      # Move EPC into $k1
addiu $v0, $zero, 0x44
slt $v0, $v0, $k0  # Ignore interrupts
bgtz $v0, _restore
mov $a0, $k0       # Move Cause into $a0
mov $a1, $k1       # EPC into $a1
jal print_excp     # Print exception error msg

_restore:        lw $a0, save0
    lw $a1, savel
    lw $k0, -4($k1)     # $k0 = previous instruction
    srl $k0, $k0, 26    # $k0 = opcode of prev instr
    ori $k1, $zero, 2   # opcode of j
    beq $k0, $k1, _delayslot #
    ori $k0, $zero, 4   # opcode of beq
    beq $k0, $k1, _delayslot
    # and so on for: jr, jal, bne, bltz, bgezal, bczt...

_done:          mfc0 $k1, $14      # reload EPC into $k1
    addiu $k1, $k1, 4   # Do not reexecute fault instr
jr $k1
rfe

_delayslot:    mfc0 $k1, $14      # reload EPC into $k1
    addiu $k0, $k1, -4  # $k0 = EPC - 4
    addiu $k1, $k1, -4  # $k1 = EPC + 4
jr $k0
rfe

.kdata
save0:       .word 0
savel:       .word 0
```

This problem is hard. The basic idea of this solution is to do everything possible in order not to touch the instruction that caused the exception. We need a way to “poke” the branching instruction, that is, execute the instruction without executing any instructions around it. This procedure works by calling the branching instruction with \( jr \), but putting a \( j \) in the delay-slot of the \( jr \), so that we will jump back after executing the branching instruction and not execute its regular delay-slot. If it turns out that the branch is not taken (which may happen with a \( bne \) or \( beq \)), then we jump back to EPC+4.

Note (1) To be thoroughly clear, no pseudoinstructions are used in this solution.
Note (2) There are many bugs in the interrupt handler on page A-35 of P&H second edition, such as not saving $r3 and an incorrect interrupt mask of 0x44. In fact, on a final exam in CS 61C taught by David Patterson in spring 1999, an entire exam question was on locating the bugs in the code from page A-35. The solution given here copies most of the code from page A-35 and doesn’t bother fixing the bugs.

Note (3) What happens if you jump into a branch delay slot? In this case, just rolling back won’t work. The answer to this is that this is against MIPS convention, so no compilers would produce code where you jump into a branch delay slot.

Note (4) If someone could find a cleaner solution to this problem, it would be greatly appreciated!