Solutions for Chapter 1 Exercises

1.1 5, CPU

1.2 1, abstraction

1.3 3, bit

1.4 8, computer family

1.5 19, memory

1.6 10, datapath

1.7 9, control

1.8 11, desktop (personal computer)

1.9 15, embedded system

1.10 22, server

1.11 18, LAN

1.12 27, WAN

1.13 23, supercomputer

1.14 14, DRAM

1.15 13, defect

1.16 6, chip

1.17 24, transistor

1.18 12, DVD

1.19 28, yield

1.20 2, assembler

1.21 20, operating system

1.22 7, compiler

1.23 25, VLSI

1.24 16, instruction

1.25 4, cache •

1.26 17, instruction set architecture

1.27 21, semiconductor

1.28 26, wafer

1.29 i

1.30 b

1.31 e

1.32 i

1.33 h

1.34 d

1.35 f

1.36 b

1.37 c

1.38 f

1.39 d

1.40 a

1.41 c

1.42 i

1.43 e

1.44 g

1.45 a

1.46 Magnetic disk:

Time for 1/2 revolution =1/2 rev x 1/7200 minutes/rev X 60 seconds/ minutes^3 4.17 ms

Time for 1/2 revolution = 1/2 rev x 1/10,000 minutes/rev X 60 seconds/ minutes = 3 ms

1.47 (DVD):

Bytes on center circle = 1.35 MB/seconds X 1/1600 minutes/rev x 60 seconds/minutes = 50.6 KB

Bytes on outside circle = 1.35 MB/seconds X 1/570 minutes/rev X 60 seconds/minutes = 142.1 KB

1.48 Total requests bandwidth = 30 requests/sec X 512 Kbit/request = 15,360 Kbit/sec < 100 Mbit/sec. Therefore, a 100 Mbit Ethernet link will be sufficient.

1.49 Possible solutions:

Ethernet, IEEE 802.3, twisted pair cable, 10/100 Mbit

Wireless Ethernet, IEEE 802.1 lb, no medium, 11 Mbit

Dialup, phone lines, 56 Kbps

ADSL, phone lines, 1.5 Mbps

Cable modem, cable, 2 Mbps

1.50

a. Propagation delay = *mis* sec

Transmission time = *LIR* sec

End-to-end delay =*m/s+L/R*

b. End-to-end delay =*mls+ LJR+t*

c. End-to-end delay = *mis* + 2I/R + f/2

1.51 Cost per die = Cost per wafer/(Dies per wafer x Yield) = 6000/(1500 x 50%) = 8

Cost per chip = (Cost per die + Cost_packaging + Cost_testing)/Test yield =

(8 + 10)/90% = 20

Price = Cost per chip x (1 + 40%) - 28

If we need to sell *n* chips, then 500,000 + 20« = 28», *n* = 62,500.

1.52 CISCtime = P x 8 r = 8 P r n s

RISC time = 2Px 2T= 4 *PTns*

RISC time = CISC time/2, so the RISC architecture has better performance.

1.53 Using a Hub:

Bandwidth that the other four computers consume = 2 Mbps x 4 = 8 Mbps

Bandwidth left for you = 10 - 8 = 2 Mbps

Time needed = (10 MB x 8 bits/byte) / 2 Mbps = 40 seconds

Using a Switch:

Bandwidth that the other four computers consume = 2 Mbps x 4 = 8 Mbps

Bandwidth left for you = 10 Mbps. The communication between the other computers will not disturb you!

Time needed = (10 MB x 8 bits/byte)/10 Mbps = 8 seconds

1.54 To calculate d = a x f c - a x c, the CPU will perform 2 multiplications and 1 subtraction.

Time needed = 1 0 x 2 + 1 x 1 = 2 1 nanoseconds.

We can simply rewrite the equation *&sd = axb-axc= ax (b-c)*. Then 1 multiplication and 1 subtraction will be performed.

Time needed = 1 0 x 1 + 1 x 1 = 11 nanoseconds.

1.55 No solution provided.

1.56 No solution provided.

1.57 No solution provided.

1.68 Performance characteristics:

Network address

Bandwidth (how fast can data be transferred?)

Latency (time between a request/response pair)

Max transmission unit (the maximum number of data that can be transmitted in one shot)

Functions the interface provides:

Send data

Receive data

Status report (whether the cable is connected, etc?)

1.69 We can write Dies per wafer = /((Die area)"^{1}) and Yield = /((Die area)"^{2}) and thus Cost per die = /((Die area)^3).

1.60 No solution provided.

1.61 From the caption in Figure 1.15, we have 165 dies at 100% yield. If the defect density is 1 per square centimeter, then the yield is approximated by

$$\frac{1}{\left[1+\left(\frac{\frac{1}{100\text{mm}^2}\times 250\text{mm}^2}{2}\right)\right]^2} = .198.$$

Thus, 165 x .198 = 32 dies with a cost of $1000/32 = $31.25 per die.

1.62 Defects per area.

1.63 $$\text{Yield} = \frac{1}{(1 + \text{Defects per area} \times \text{Die area}/2)^2}$$

$$\text{Defects per area} = \frac{2}{\text{Die area}}\left(\frac{1}{\sqrt{\text{Yield}}} - 1\right)$$

1.64

1980	Die ares	0.16
	Yield	0.48
	Defect density	5.54
1992	Die area	0.97
	Yield	0.48
	Defect density	0.91
1992 + 19S0	improvement	6.09

Solutions for Chapter 2 Exercises

2.2 By lookup using the table in Figure 2.5 on page 62,

7ffififfo$_{\text{hoi}}$ = 0111 1111 1111 1111 1111 1111 1111 1010$_{\text{two}}$

= 2,147,483,642^.

2.3 By lookup using the table in Figure 2.5 on page 62,

1100 1010 1111 1110 1111 1010 1100 111<U, = cafe face$_{\text{hex}}$.

2.4 Since MIPS includes add immediate and since immediates can be positive or negative, subtract immediate would be redundant.

2.6

```
sll  $tO, $t3, 9    # shift $t3 left by 9, store in $tO
srl  $tO, ttO, 15   # shift $tO right by 15
```

2.8 One way to implement the code in MIPS:

```
sll     tsO, $sl, 11       # shift receiver left by 22, store in data
srl     $sO, $so, 24       # shift data right by 24 (data - receiver, receivedByte)
andi    $sl, $sl, Oxfffe   # recei ver . ready - 0:
on*     $sl, tsl, 0x0002   # recei ver.enabl e - 1;
```

Another way:

```
srl   $s0. $sl, 2       ii data = recei ver. recei vedByte
andi  $s0, $s0, 0x00ff
andi  $sl. $sl. Oxfffe  it recei ver. ready - 0;
ori   $sl, Ssl, 0x0002  it receiver.enable = 1;
```

2.9

```
1b   tsO, 0($sl)                # load the lower 8 bytes of a into bits
sll  $t0, JsO, 8                it $t0 - bits << 8
or   $s0, $s0, $tO              # bits.datal = bits.dataO
lu1  $sO, 0000 0000 OHO 0100    # bits.data? - 'd'
lui  $t0, 0000 0001 0000 0000   # load a 1 into the upper bits of $t0
or   $s0. $s0, $t0              # bits.valid - 1
```

2.10

```
      sit St3. Ss5, $zero        # test k < 0
      bne Jt3. $zero, Exit       t if so, exit
      sit St3, $s5, $t2          i test k < 4
      beq $t3, $zero, Exit       t if not, exit
      sll Jtl, Ss5, 2            § $tl - 4*k
      add $tl, «tl. It4          t $tl - SJumpTabletk)
      lw «to, 0($tl)             t $tO - JumpTable[k]
      jr $to                     t jump register
LO:   add $sO.• fs3.$s4          t k — 0
      j Exit                     # break
LI:   add $sO. fsl, $S2          t k — 1
      i Exit                     1 break
L2:   sub $sO. tsl, Js2          f k — 2
      3 Exit                     f break
L3:   sub $S0. fs3, $s4          t k — 3
      i Exit                     f break
Exit:
```

2.11

a.

```
if (k—0) f - i + j;
else if (k—1) f - g + h;
else if (k—2) f - g - h;
else if (k—3) f - i - j:
```

b.

```
        bne   $s5, $0, Cl       # branch k != 0
        add   JsO, $s3, $s4     # f - 1 + j
        j      Exit              # break
    Cl: addi  $tO, $s5, -1      # $tO - k - 1
        bne   StO, tO. C2       # branch k !- 1
        add   tsO. $sl. $s2     # f - g + h
        j      Exit              # break
    C2: addi  $tO, $s5, -I      # $tO - k - 2
        bne   $tO, $0, C3       # branch k != 2
        sub   tsO, tsl, Ss2     # f - g - h
        j      Exit              # break
    C3: addi  StO, $s5, -3      # $tO - k - 3
        bne   $tO, $0, Exit     \\ branch k != 3
        sub   $sO, $s3, $s4     # f - 1 - j
    Exit:
```

c The MIPS code from the previous problem would yield the following results:

(5 arithmetic) 1.0 + (1 data transfer) 1.4 + (2 conditional branch) 1.7
+ (2 jump) 1.2 = 12.2 cycles

while the MIPS code from this problem would yield the following:

(4 arithmetic) 1.0 + (0 data transfer)1.4 + (4 conditional branch) 1.7
+ (0jump)1.2 = 10.8 cycles

2.12 The technique of using jump tables produces MIPS code that is independent of N, and always takes the following number of cycles:

(5 arithmetic) 1.0 + (1 data transfer) 1.4 + (2 conditional branch) 1.7
+ (2 jump) 1.2= 12.2 cycles

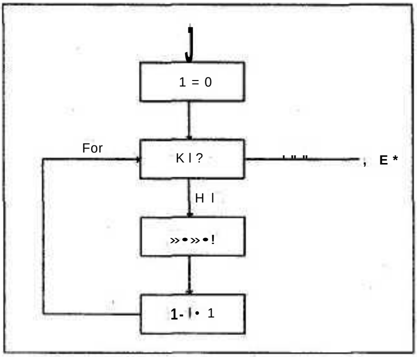
However, using chained conditional jumps takes the following number of cycles in a worst-case scenario:

(Narithmetic)1.0+ (0datatransfer)1.4 +{Nconditionalbranch)1.7
+ (0jump)1.2 = 2.7Ncycles

Hence, jump tables are faster for the following condition:

N> 12.2/2.7 = 5 *case* statements

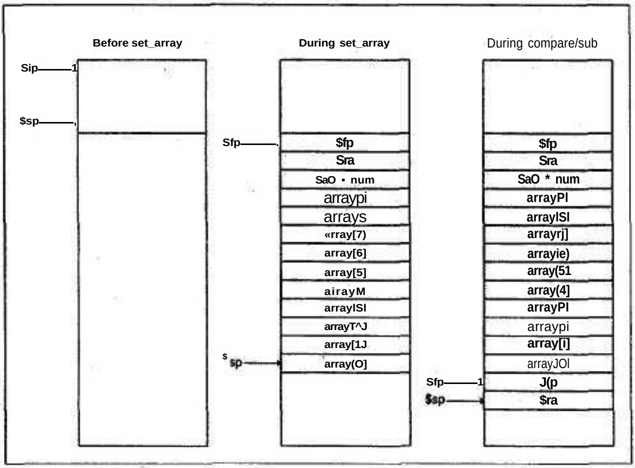
2.13



2.16 Hence, the results from using *if-else* statements are better.

```
set_array:  add!  $sp, $sp. -52      # move stack pointer
            sw    »fp. 48<$sp)       # save frame pointer
            sw    $ra, 44(tsp)       # save return address
            sw    *a0, 40($sp)       # save parameter (num)
            addi  Jfp, $sp, 48       # establish frame pointer

            add   $SO. $zero, $zero  # 1 - 0
            addi  $to, $zero, 10     # max iterations is 10
loop:       sll   $tl. $sO, 2        # $tl - i * 4
            add   $t2, Jsp, $tl      # $t2 - address of array[i]
            add   «a0, $a0, $zero    # pass num as parameter
            add   tal, $s0, $zero    # pass i as parameter
            Jal   compare            # cal1 comparedium, i)
            sw    $V0. 0($t2)        # array[i] - compare(num, i);
            addi  $sO, ISO, 1
            bne   $sO, $t0, loop     # loop if K10

            Iw    $aO, 40($sp)       # restore parameter (num)
            Iw    Sra, 44($sp)       # restore return address
            lw    $fp. 48($sp)       # restore frame pointer
            addi  $sp. $sp, 52       # restore stack pointer
            jr    (ra                # return

compare:    addi  tsp. Jsp, "8       # move stack pointer
            sw    (fp, 4(Ssp)        # save frame pointer
            sw    Jra, 0($sp)        it save return address
            addi  tfp. $sp, 4        # establish frame pointer

            jal sub                  # can jump directly to sub
            sit   $vO, $vO, $zero    # if sub(a.b) >= 0, return 1
            slti  $v0, $v0, 1

            lw    $ra, 0($sp)        # restore return address
            lw    $fp, 4($sp)        # restore frame pointer
            addi  $sp, $sp, 8        # restore stack pointer
            jr  $ra                  # return

sub:       sub    $v0, $a0, $al      # return a-b
            jr  $ra                  # return
```

The following is a diagram of the status of the stack:



2.16

```
# Description: Computes the Fibonacci function using a recursive process.
# Function:       F(n) = 0 .  if n - 0;
t                 1.        if n - 1;
#                 F(n-l) + F(n-2). otherwise.
# Input: n. which must be a nonnegative integer.
# Output:         F(n).
ii Preconditions:  none
# Instructions: Load and run the program in SPIM, and answer the prompt.
```

```
if Algorithm for main program:
#    print prompt
if    call fib(read) and print result.
# Register usage:
if  taO - n (passed directly to fib)
#  $sl - f(n)
         .data
         .align 2
if Data for prompts and output description
prmptl:  .asciiz "\n\nThis program computes the Fibonacci function."
prmpt2:  .asciiz "\nEnter value for n: "
descr:   .asciiz "fib(n) - "
         .text
         .align 2
      •  -globl___start
_start:
if Print the prompts
      li  $vO, 4          if print_str system service ...
      la  $aO, prmptl     # ... passing address of first prompt
      syscal1
      li  SvO, 4          # print_str system service ...
      la  $aO, prmpt2     if ... passing address of 2nd prompt
      syscal1
if Read n and call fib with result
      li  $vO, 5          if read_int system service
      syscall
      move $aO, $vO       if $aO - n = result of read
      jal fib             § call fib(n)
      move $sl, $vO       if $sl = fib(n)
# Print result
      li $v0, 4           if print_str system service ...
      la $a0, descr       it ... passing address of output descriptor
      syscall
      li $v0, 1           if print_int system service ...
      move $a0, $sl       it ... passing argument fib(n)
      syscall
if Call system - exit
      li $v0. 10
      syscal1
if Algorithm for Fib(n):
it   if (n == 0) return 0
if   else if (n — 1) return 1
#    else return fib(n-l) + f1b(n-2).
it
```

```
# Register usage:
#  $aO - n (argument)
#  $tl - fibCn-1)
#  $t2 - fibCn-2)
#  $vO = 1 (for comparison)
#
# Stack usage:
# 1.  push return address, n, before calling fib(n-l)
# 2.  pop n
# 3.  push n, fib(n-l), before calling fibtn-2)
# 4.  pop fib(n-l), n, return address
fib: bne $aO, $zero, fibneO  # if n ~ 0 ...
      move $vO, $zero         # ... return 0
      jr $31
fibneO:                       # Assert: n !- 0
      li tvO, 1
      bne $aO, $vO, fibnel    # if n — 1 ...
      jr $31                  # ... return 1
fibnel:                       # Assert: n > 1
## Compute fib(n-l)
      addi $sp, $sp, -8       # push ...
      sw $ra, 4($sp)          # ... return address
      sw $aO, O($sp)          # ... and n
      addi $aO, $aO, -1       # pass argument n-1 ...
      jal fib              # ... to fib
      move $tl, $vO           # $tl = fib(n-l)
      lw $aO, O($sp)          # pop n
      addi $sp, $sp, 4        # ... from stack
## Compute fib(n-2)
      addi $sp, $sp, -8    tf    push ...
      sw $aO, 4($sp)          # ... n
      sw $tl, 0($sp)          # ... and fib(n-l)
      addi $aO, $aO, -2       # pass argument n-2 ...
      jal fib              # ... to fib
      move $t2, $vO           # tt2 = fib(n~2)
      lw $tl, OC$sp)          # pop fib(n-l) ...
      Iw $aO, 4{$sp)          # ... n
      lw $ra, 8{$sp)          # ... and return address
      addi $sp, $sp, 12       # ... from stack
## Return fib(n-l) + ffbCn-2)
      add $vO, $tl. $t2       # $vO - fib(n) = fib(n-l) + fib(n-2)
      jr $31                  # return to caller
```

2.17

```
# Description:   Computes the Fibonacci function using an
it               iterative process.
# Function:     F(n) =0 ,   if n = 0;
#                  1,     1f n - 1;
#                   F(n-l) + Ftn-2). otherwise.
it Input:        n, which must be a nonnegative integer.
it Output:      F(n).
# Preconditions: none
# Instructions:  Load and run the program in SPIH, and answer
it               the prompt.
it

# Algorithm for main program:
it    print prompt
it    call fib(l, 0, read) and print result.
it
# Register usage:
#  $a2 - n (passed directly to fib)
it  $sl - fCn)
                .data
                .align 2
# Data for prompts and output description
prmptl:          .asciiz "\n\nThis program computes the the
                         Fibonacci functi on."
prmpt2:         .asciiz "\nEnter value for n: "
descr:          .asciiz "fib{n) - "
                .text
                .align 2
                .globi___start
—start:
it Print the prompts
                li $vo, 4         # print_str system service ...
                1a $aO, prmptl    # ... passing address of first
                                    prompt
                syscal1
                li $vo, 4         # print_str system service ...
                la $aO, prmpt2    # ... passing address of 2nd
                prompt syscall
# Read n and ca1 fib with result
                li $v0, 5         # read_int system service
                syscal1
                move $a2, $vO     # $a2 - n - result of read
                li $al, 0         # Sal - fib(O)
                li $a0, 1         it $aO - fibtl)
                jal fib           it call fib(n)
                move Isl, IvO     it $sl - fib(n)
```

```
it Print result
                11 JvO, 4          it print_str system service ...
                la iaO, descr      it ... passing address of output
                                   it descriptor
                syscal1
                If $vO, 1          it print_int system service ...
                move $aO, ts1      it ... passing argument fib(n)
                syscal1
# Call system - exit
                li $vO. 10
                syscal1
# Algorithm for FibCa. b, count):
#   if (count — 0) return b
#   else return fib(a + b, a, count - 1).
it
it Register usage:
it  $aO - a - fib(n-l)
it  Sal - b - fib{n-2)
it  $a2 - count (initially n, finally 0).
it  ttl = temporary a + b
fib:            bne $a2, $zero. fibneO          # if count — 0 ...
                move $vO, $al                   # ... return b
                jr $31
fibneO:                                         # Assert: n !- 0
                addi $a2, $a2, -1               # count - count - 1
                add $tl, $aO, $ai               # $tl - a + b
                move $al, taO                   it b = a
                move $aO, ttl                   # a - a + old b
                j fib                           it tail call fib(a+b. a, count-1)
```

2.18 No solution provided.

2.19 Irisin ASCII: 73 114 105 115

Iris in Unicode: 0049 0072 0069 0073

Julie in ASCII: 74 117 108 105 101

Julie in Unicode: 004A 0075 006C 0069 0065

2.20 Figure 2.21 shows decimal values corresponding to ACSII characters.

A		b	y	t	*e*		i	s		8		b	i	t	S	
65	32	98	121	116	101	32	101	115	32	56	32	98	101	116	115	0

2.29

```
         add  $to,  Szer0. $zero      # initialize running sum StO - 0
1 oop:   beq  $al.  Sier0, finish     # finished when Sal is 0
         add  StO.  StO, SaO          # compute running sum of $aO
         sub  $al,  Sal, 1            # compute this $al times
         j    loop
finish:  addi StO.  StO, 100          4 add 100 to a * b
         add  SvO,  StO, Szero        # return a * b + 100
```

The program computes a * b + 100.

2.30

```
        sll  Sa2. $a2. 2            # max i- 2500 * 4
        sll  Sa3. 8a3, 2            # max j- 2500 * 4
        add  SvO. Szero , Szero     # tvO - 0
        add  StO. Szero . Szero     # 1 - 0
outer:  add  St4, Sao, StO          # $t4 = address of array l[i]
        lw   $t4, 0(St41            # $t4 - array l[i]
        add  »tl, Szero . Szero     # j - 0
Inner:  add  St3. Sal, Stl          # $t3 - address of array 2[J]
        lw   St3, 0(St3)            # $t3 - array 2[J]
        bne  »t3. St4, skip         # if (array l[i] !- array 2[j]) skip $v0++
        addi SvO, SvO, 1            # $v0++
skip    addi Stl, Stl, 4            # j++
        bne  m.  Sa3, inner         # loop if j I- 2500 * 4
        addi StO, StO, 4            # i++
        bne  StO. Sa2. outer        # loop 1f 1 !- 2500 * 4
```

The code determines the number of matching elements between the two arrays and returns this number in register $v0.

2.31 Ignoring the four instructions before the loops, we see that the outer loop (which iterates 2500 times) has three instructions before the inner loop and two after. The cycles needed to execute these are 1 + 2 + 1 = 4 cycles and 1 + 2 = 3 cycles, for a total of 7 cycles per iteration, or 2500 x 7 cycles. The inner loop requires 1 + 2 + 2 + 1 + 1 + 2 = 9 cycles per iteration and it repeats 2500 x 2500 times, for a total of 9 x 2500 x 2500 cycles. The total number of cycles executed is therefore (2500 x 7) + (9 x 2500 x 2500) = 56,267,500. The overall execution time is therefore (56,267,500) / (2 x 10^9) = 28 ms. Note that the execution time for the inner loop is really the only code of significance.

2.32 ori H I , $t0. 25 # register ttl - St0 I 25;

2.34

```
      addi $v0, $zero,  -1    # Initialize to avoid counting zero word
loop: lw,  $vl, 0($a0)    tf    Read next word from source
      addi $v0, $v0, 1        # Increment count words copied
      sw   $vl, 0($al)        # Write to destination
      addi $a0, $a0, 4        # Advance pointer to next source
      addi Sal, $al, 4        # Advance pointer to next destination
      bne  $vl, tzero, loop  # Loop if word copied != zero
```

Bug I:Count($vO) is initialized to zero, not-1 to avoid counting zero word.

Bug 2: Count (SvO) is not incremented.

Bug 3: Loops if word copied is equal to zero rather than not equal.

2.37

Pseudoinstruction	What it accomplishes	Solution
move $t1, $t2	$t1 = $t2	add $t1, $t2, $zero
clear- ItO	UO-0	add tt0, $zero. tzero
beq ttl. small. L	if<*tl = small)goto L	li $at, small beq ttl, $at. L
beq H2. big. L	if <tt2 ⟹ big) go to L	11 1at, big beq tat, $zero. L
11 ttl. small	ttl* small	addi tt1. $zero. small
11 JtZ, big	ttZ = big	lui %t2, upper(big) on" It2, $t2. lower(big)
ble tt3. **St5.** L	tf<*t3 <=It5}goto L	sit tat, $t5, tt3 beq tat, $zero, L
bgt $t4, $t5, L	If{tt4>it5)gotoL	sit tat, $t5. **St4** bne $at, $zero. L
bge tt5. tt3. L	If(tt5>=tt3)gotoL	sit 1at, $t5. tt3 beq Iat, $zero, L
addi ttO. ttZ. big	StO = ttZ + big	11 tat, big add tt0, $t*1.* tat
lw it5, b1g(Jt2)	tt5 = Memorylt2 + big]	li $at, big add Jat. $at. ***%xz*** lw $t5, $t*2.* tat

Note: In the solutions, we make use of the *1* i instruction, which should be implemented as shown in rows 5 and 6.

2.38 The problem is that we are using PC-relative addressing, so if that address is too far away, we won't be able to use 16 bits to describe where it is relative to the PC. One simple solution would be

```
here: bne    $sO, $s2, skip
      j       there
skip:
      ...
there: add   $sO, $sO, $sO
```

This will work as long as our program does not cross the 256MB address boundary described in the elaboration on page 98.

2.42 Compilation times and run times will vary widely across machines, but in general you should find that compilation time is greater when compiling with optimizations and that run time is greater for programs that are compiled without optimizations.

2.45 Let /be the number of instructions taken on the unmodified MIPS. This decomposes into 0.42/arithmetic instructions (24% arithmetic and 18% logical), 0.361 data transfer instructions, 0.18/conditional branches, and 0.031 jumps. Using the CPIs given for each instruction class, we get a total of (0.42 x 1.0 + 0.36 x 1.4 + 0.18 x 1.7 + 0.03 x 1.2) x /cycles; if we call the unmodified machine's cycle time Cseconds, then the time taken on the unmodified machine is (0.42 x 1.0 + 0.36 x 1.4 + 0.18 x 1.7 + 0.03 x 1.2) x /x Cseconds. Changing some fraction,/ (namely 0.25) of the data transfer instructions into the autoincrement or autodecrement version will leave the number of cycles spent on data transfer instructions unchanged. However, each of the 0.36 x / x /data transfer instructions that are changed corresponds to an arithmetic instruction that can be eliminated. So, there are now only (0.42- (036 *xf))* x *I* arithmetic instructions, and the modified machine, with its cycle time of 1.1 x Cseconds, will take {(0.42 - 0.36/) x 1.0 + 0.36 x 1.4 + 0.18 x 1.7 + 0.03 x 1.2) x I x 1.1 x Cseconds to execute. When/is 0.25, the unmodified machine is 2.2% faster than the modified one.

2.46 Code befot^{m}e:

```
         In    m.   4(Ss6)                          # temp reg $t2 - length of array save
Loop:    sit   st0. Ss3, Szero                      # temp reg $tO - 1 if 1 < 0
         bne   st0, Szero . IndexOutOfBounds        tt if 1< 0, goto Error
         sit   st0. Ss3. St2                        # temp reg $t0 = 0 if i >= length
         beq   st0. Szero, IndexOutOfBounds         # if i >- length, goto Error
         sll   Stl, Ss3, 2                          # temp reg $tl = 4 * i
         add   Stl. Stl. $S6                        # Stl - address of saved]
         Iw    st0, 8($tl)                          # temp reg $t0 = save[i]
         bne   st0, Ss5. Exit                       # go to Exit if save[i] !* k
         addi  Ss3, Ss3, 1                          # i - 1 + 1
         1     Loop
Exit:
```

The number of instructions executed over 10 iterations of the loop is 10 x 10 + 8 + 1 = 109. This corresponds to 10 complete iterations of the loop, plus a final pass that goes to Exit from the final bne instruction, plus the initial Iw instruction. Optimizing to use at most one branch or jump in the loop in addition to using only at most one branch or jump for out-of-bounds checking yields:

Code after:

```
       Iw   UZ. 4($s6)                    # temp reg $t2 = length of array save
       sit  tto, $S3, tzero               # temp reg $tO - 1 if i < 0
       sit  tt3, $S3, $tz                 # temp reg $t3 - 0 if i >- length
       slti tt3. $t3, 1                   # flip the value of $t3
       or   (t3. >t3, tto                 # $t3 - 1 if i is out of bounds
       bne  tt3. (zero, IndexOutOfBounds  # if out of bounds, goto Error
       stl  ttl. »s3, 2                   # tem reg Stl - 4 * 1
       add  ttl. ttl, ts6                 # Stl - address of saved]
       In   tto, 8(ttl)                   # temp reg $tO - saved]
       bne  sto, ts5, Exit                # go to Exit if save[i] !- k
Loop:  addi ts3. *s3, 1                   # 1 - 1 + 1
       sit  tto. $S3, tzero               # temp reg $tO = 1 if i < 0
       sit  tt3. >s3. tt2                 # temp reg St3 = 0 if i >- length
       slti St3, «t3. 1                   # flip the value of $t3
       or   $t3. tt3, tto                 # $t3 = 1 if i is out of bounds
       bne  it3, tzero, IndexOutOfBounds •# if out of bounds, goto Error
       addi itl. ttl, 4                   # temp reg $tl = address of saved]
       lu   tto. 8($tl)                   # temp reg $tO = save[i]
       beq  no. «s5. Loop                 # go to Loop if save[i] = k
Exit:
```

The number of instructions executed by this new form of the loop is 10+10*9 = 100.

2.47 To test for loop termination, the constant 401 is needed. Assume that it is placed in memory when the program is loaded:

```
       lw    tt8, AddressConstant401(tzero) it tt8 - 401
       lw    tt7, 4(taO)                    it tt7 = length of a[]
       lw    tt6, 4(tal)                    It St6 - length of b[]
       add   tto. tzero, tzero              it Initialize 1 - 0
Loop:  sit   $t4. ttO. tzero                it $t4 - 1 If 1 < 0
       bne   tt4. tzero, IndexOutOfBounds   it if i< 0. goto Error
       sit   tt4, ttO. St6                  it tt4 - 0 If 1 >- length
       beq   tt4. Jzero, IndexOutOfBounds   it if i >- length, goto Error
       sit   $t4. ttO, tt7                  it tt4 = 0 if i >- length
       beq   tt4, tzero, IndexOutOfBounds   it if i >- length, goto Error
       add   ttl, tal, StO                  it ttl - address of b[i]
       lw    tt2. 8(Stl)                    it St2 - bti]
       add   $t2. tt2, tsO                  it $t2 - b[i] + c
       add   $t3. taO. ttO                  it tt3 - address of a[i]
       sw    tt2, 8(tt3)                    it a[i] - b[i] + c
       addi  no, ttO, 4                     it i - i + 4
       sit   tt4. StO, St8                  it tt8 - 1 If ttO < 401, i.e., i <= 100
       bne   tt4. tzero, Loop               it goto Loop if i <= 100
```

The number of instructions executed is 4 + 101 X 14= 1418. The number of data references made is 3 + 101 X 2 = 205.

2.48

```
compareTo: sub $v0, $a0, Sal    # return v[i].value - v[j+l],value
           jr  $ra              # return from subroutine
```

2.49 From Figure 2.44 on page 141, 36% of all instructions for SPEC2000int are data access instructions. Thus, for every 100 instructions there are 36 data accesses, yielding a total of 136 memory accesses (1 to read each instruction and 36 to access data).

a. The percentage of all memory accesses that are for data = 36/136 = 26%.

b. Assuming two-thirds of data transfers are loads, the percentage of all memory accesses that are reads = (100 + (36 x 2/3)}/136 = 91%.

2.50 From Figure 2.44,39% of all instructions for SPEC2000fp are data access instructions. Thus, for every 100 instructions there are 39 data accesses, yielding a total of 139 memory accesses (1 to read each instruction and 39 to access data).

a. The percentage of all memory accesses that are for data = 39/139 = 28%.

b. Assuming two-thirds of data transfers are loads, the percentage of all memory accesses that are reads = (100 + (39 x 2/3))/139 = 91%.

2.51 Effective CPI = Sum of (CPI of instruction type x Frequency of execution)

The average instruction frequencies for SPEC2000int and SPEC2000fp are 0.47 arithmetic (0.36 arithmetic and 0.11 logical), 0.375 data transfer, 0.12 conditional branch, 0.015 jump. Thus, the effective CPI is 0.47 x 1.0 + 0.375 x 1.4 + 0.12 x 1.7 + 0.015x1.2=1.2.

2.52

Accumulator		
Instruction	**Code bytes**	**Data bytes**
load b # Acc - b;	3	4
add c # Acc +- c;	3	4
store a # a = Acc;	3	4
add c # Acc +- c:	3	4
store b # Acc - b;	3	4
neg # Acc -- Ace;	1	0
add a. # Acc -- b;	3	4
store d # d - ACC;	3	4
Total:	22	28

Code size is 22 bytes, and memory bandwidth is 22 + 28 = 50 bytes.

Stack		
Instruction	**Code bytes**	**Data bytes**
push b	3	4
push c	3	4
add	1	0
dup	1	0

Stack		
Instruction	Code bytes	Data bytes
pop a	3	4
push c	3	4
add	1	0
duo	1	0
pop b	3	4
neg	1	0
push a	3	4
add	1	0
pop d	3	4
Total:	27	28

Code size is 27 bytes, and memory bandwidth is 27 + 28 = 55 bytes.

Memory-Memory		
Instruction	Code bytes	Data bytes
add a, b, c # a=b+c	7	12
I add b, a, c # b-a+c	7	12
sub d, a, b # d=a-b	7	12
Total:	21	36

Code size is 21 bytes, and memory bandwidth is 21 + 36 = 57 bytes.

Load-Store		
Instruction	Code bytes	Data bytes
load $1, b # $1 = b;	4	4
load I2, c # $2 = c;	4	4
add $3. $1, 12 # $3 = $i + SZ	3	0
store S3. a # a - i3;	4	4
add S1. 12. $3 # $1 = $2 + $3;	3	0
store $1. b # b - $1;	4	4
sub $4, 13, tl # $4 = $3 - tl:	3	0
store $4. d # d - t4;	4	4
Total:	29	20

Code size is 29 bytes, and memory bandwidth is 29 + 20 = 49 bytes.

The load-store machine has the lowest amount of data traffic. It has enough registers that it only needs to read and write each memory location once. On the other hand, since all ALU operations must be separate from loads and stores, and all operations must specify three registers or one register and one address, the load-store has the worst code size. The memory-memory machine, on the other hand, is at the other extreme. It has the fewest instructions (though also the largest number of bytes per instruction) and the largest number of data accesses.

2.53 To know the typical number of memory addresses per instruction, the nature of a typical instruction must be agreed upon. For the purpose of categorizing computers as 0-, 1-, 2-, 3-address machines, an instruction that takes two operands and produces a result, for example, a d d, is traditionally taken as typical.

Accumulator: An add on this architecture reads one operand from memory, one from the accumulator, and writes the result in the accumulator. Only the location of the operand in memory need be specified by the instruction. Category: 1-address architecture.

Memory-memory: Both operands are read from memory and the result is written to memory, and all locations must be specified. Category: 3-address architecture.

Stack: Both operands are read (removed) from the stack (top of stack and next to top of stack), and the result is written to the stack (at the new top of stack). All locations are known; none need be specified. Category: 0-address architecture.

Load-store: Both operands are read from registers and the result is written to a register. Just like memory-memory, all locations must be specified; however, location addresses are much smaller—5 bits for a location in a typical register file versus 32 bits for a location in a common memory. Category: 3-address architecture.

2.54

```
        sbn temp, temp, .+1   # clears temp, always goes to next instruction
start:  sbn temp, b, .+1      # Sets temp = -b
        sbn a, temp, .+1      # Sets a - a - temp - a - {-b) - a + b
```

2.55 There are a number of ways to do this, but this is perhaps the most concise and elegant:

```
        sbn  c, c, .+1        # c = 0;
        sbn  tmp, tmp, .+1    # tmp - 0;
loop:   sbn  b, one, end      # while {--b >= 0)
        sbn  tmp, a, loop     # tmp -= a; /* always continue */
end:    sbn  c, tmp, .+1      # c = -tmp; /* - a x b */
```

2.56 Without a stored program, the programmer must physically configure the machine to run the desired program. Hence, a nonstored-program machine is one where the machine must essentially be rewired to run the program. The problem with such a machine is that much time must be devoted to reprogramming the machine if one wants to either run another program or fix bugs in the current program. The stored-program concept is important in that a programmer can quickly modify and execute a stored program, resulting in the machine being more of a general-purpose computer instead of a specifically wired calculator.

2.57

MIPS:

```
       add    tto.  tze ro, $zero    t 1 - 0
       addi   ttl,  tze ro, 10       t set max iterations of loop
loop:  sll    $t2.  to,  2           t tt2 - i * 4
       add    $t3,  tt2 , tal        1 tt3 - address of b[i]
       Iw     tt4,  0(tt3)           t tt4 - b[i]
       add    tt4.  tt4 , tto        t tt4 - bCi] + i
       sll    $t2,  to,  4           t tt2 - 1 * 4 * 2
       add    $t3,  tt2 , taO        t tt3 - address of a[2i]
       sw     tt4,  0(tt3)           t a[2i] - b[i] + 1
       addi   (to,  sto, 1           t i++
       bne    $to.  ttl . loop       t loop if i !- 10
```

PowerPC:

```
       add    $to,  tze ro, tzero    t i --0
       addi   $tl,  tzero, 10        # set max iterations of loop
loop:  1 wu   tt4,  4(t al)          # tt4 = bti]
       add    tt4,  tt4 , tto        # tt4 - bti] + 1
       sll    tt2,  to,  4           t tt2 - 1 * 4 * 2
       sw     ft4,  taO +tt2         1 a[2i] - b[i] + i
       addi   tto.  $to, 1           # i++
       bne    tto,  $tl , 1 oop      II oop if i !- 10
```

2.58

MIPS:

```
        add   tv0, $zero, $zero     t freq = 0
        add   $t0, $zero, Szero     t i - 0
        addi  St8, Szero, 400       t St8 - 400
outer:  add   St4, $aO, StO         t St4 - address of a[i]
        1u    St4, 0($t4)           itSt4 - a[i]
        add   $sO, $zero, Szero     # x - 0
        add!  $tl. $zero, 400       # j - 400
inner:  add   St3, $aO, $tl         f St3 - address of a[j]
        lw    $t3. 0($t3)           1 St3 - a[j]
        bne   St3, St4, skip        t if (a[1] !•• a[j]l skip x++
        addi  SsO, SsO, 1           t X++
skip:   addi  Stl, Stl, -4          t J--
        bne   $tl. Szero, inner     t loop if j !- 0
        sit   *t2, SsO, SvO         t St2 - 0 if x >= freq
        bne   $t2, $zero. next      # skip freq - x if
        add   $vO, SsO, Szero       § freq = x
next:   addi  St0, StO, 4           1 i++
        bne   tt0, St8, outer       1 loop if i !- 400
```

PowerPC:

```
        add   tv0, Szero, Szero     t freq - 0
        add   $t0, Szero, Szero     t 1 - 0
        addi  «t8, Szero, 400       t St8 - 400
        add   St7, SaO, Szero       t keep track of a[i] with update addressing
outer:  lwu   (t4, 4(St7)           t $t4 - a[i]
        add   SsO, Szero, Szero     t x - 0
        addi  Sctr, Szero, 100      # i - 100
        add   St6, SaO, Szero       # keep track of a[j] with update addressing
inner:  lwu   St3, 4($t6)           t St3 - a[j]
        bne   $t3. St4, skip        t if !a[i] !•- a[j]) skip x++
        addi  $sO. SsO, 1           t X++
skip:   be    inner , $ctr!-0       # j--. loop If j!-0
        sit   stz, SsO, $vO         t tt2 - 0 if x >- freq
        bne   $t2. $zero, next      # skip freq - x if
        add   $vO, SsO, $zero       t freq - x
next:   addi  no. $to, 4            t 1++
        bne   no. $t8. outer        # loop if 1 !- 400
```

2.59

```
xor $s0, $s0, $sl
xor $sl, SsO, Isl
xor SsO. SsO. $sl
```

Solutions for Chapter 3 Exercises

3.1 0000 0000 0000 0000 0001 0000 0000 0000_{two}

3.2 1111 1111 1111 1111 1111 1000 0000 0001_{two}

3.3 1111 1111 1110 0001 0111 1011 1000 0000_{two}

3.4 -250_{ten}

3.5 -17_{ttn}

3.6 2147483631_{wn}

3.7

```
        addu    $t2, Izero, $t3  # copy St3 into $t2
        bgez    $t3, next        # if $t3 >= 0 then done
        sub     tt2, Szero, St3  # negate $t3 and place into $t2
Next:
```

3.9 The problem is that A_1 ower will be sign-extended and then added to $t0. The solution is to adjust A_upper by adding 1 to it if the most significant bit of A_l ower is a 1. As an example, consider 6-bit two's complement and the address 23 = 010111. If we split it up, we notice that A_l ower is 111 and will be sign-extended to 111111 =-1 during the arithmetic calculation. A_upper_adjusted = 011000 = 24 (we added 1 to 010 and the lower bits are all Os). The calculation is then24+-l = 23.

3.10 Either the instruction sequence

```
     addu $t2, $t3, $t4
     situ $t2, $t2. $t4
```

or

```
     addu $t2, $t3, $t4
     situ $t2. -$t2, $t3
```

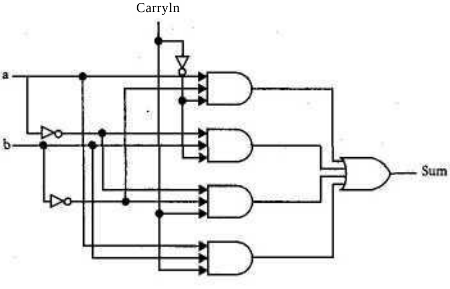
works.

3.12 To detect whether $s0 < $s1, it's tempting to subtract them and look at the sign of the result. This idea is problematic, because if the subtraction results in an overflow, an exception would occur! To overcome this, there are two possible methods: You can subtract them as unsigned numbers (which never produces an exception) and then check to see whether overflow would have occurred. This method is acceptable, but it is lengthy and does more work than necessary. An alternative would be to check signs. Overflow can occur if $s0 and (-$s1) share the same sign; that is, if $ s 0 and $ s 1 differ in sign. But in that case, we don't need to subtract them since the negative one is obviously the smaller! The solution in pseudocode would be

```
if <$s0<0) and (Ssl>0) then
      $tO:-l
else if <$s0>0) and {$sl<OJ then
      $tO:-O
else
      $tl:-$sO-Ssr
      if ($tl<0) then
            $tO:-l
      else
            $tO:-O
```

3.13 Here is the equation:

$$\text{Sum} = (a \cdot \overline{b} \cdot \overline{\text{Carryln}}) + (\overline{a} \cdot b \cdot \overline{\text{Carryln}}) + (\overline{a} \cdot \overline{b} \cdot \text{Carryln}) + (a \cdot b \cdot \text{Carryln})$$



3.23

Current bits		Prov. bits		
ai + 1	ai	ai - 1	Operation	Reason
0	0	0	None	Middle of a string of Os
0	0	1	Add the multiplicand	End of a string of Is
0	1	0	Add the multiplicand	A string of one 1, so subtract the multiplicand at position i for the beginning of the string and add twice the multiplicand (twice to align with position 1 + I) for the end of the string; net result, add the multiplicand
0	1	1	Add twice the multiplicand	End of a string of Is; must align add with 0 In position i + 1
1	0	0	Subtract twice the multiplicand	Beginning of a string of Is; must subtract with 1 in position 1 +1
1	0	1	Subtract the multiplicand	End of string of Is, so add multiplicand, plus beginning of a string of Is, so subtract twice the multiplicand; net result Is to subtract Hie multiplicand
1	1	0	Subtract the multiplicand	Beginning of a string of Is
1	1	1	None	Middle of a suing of Is

One example of 6-bit operands that run faster when Booth's algorithm looks at 3 bits at a time is 21_{ten} x 27^, = 567^.

Two-bit Booth's algorithm:

010101 $=2I_{ten}$
X011011 = 27^

- 010101 10 string (always start with padding 0 to right of LSB)
000000 11 string, middle of a string of Is, no operation
+ 010101 01 string, add multiplicand
- 010101 10 string, subtract multiplicand
000000 11 string
+ 010101 01 string

11111101011 two's complement with sign extension as needed
0000000000 zero with sign extension shown
000010101 positive multiplicand with sign extension
11101011
0000000
(-010101

01000110111 $=567_{ten}$

Don't worry about the carry out of the MSB here; with additional sign extension for the addends, the sum would correctly have an extended positive sign. Now, using the 3-bit Booth's algorithm:

```
     010101     =21ten
    X0110U      =27^
   ----------------------
    - 010101    110 string (always start with padding 0 to right of LSB)
  —010101       101 string, subtract the multiplicand
+• 0101010      011 string, add twice the multiplicand (i.e., shifted left 1 place)
 -------------
  11111101011   two's complement of multiplicand with sign extension
  111101011     two's complement of multiplicand with sign extension
+ 0101010
 ---------------------
  01000110111   "S67,,,,
```

Using the 3-bit version gives only 3 addends to sum to get the product versus 6 addends using the 2-bit algorithm.

Booth's algorithm can be extended to look at any number of bits *b* at a time. The amounts to add or subtract include all multiples of the multiplicand from 0 to $2^{*6"1'}$. Thus, for *b*> 3 this means adding or subtracting values that are other than powers of 2 multiples of the multiplicand. These values do not have a trivial "shift left by the power of2numberofbitpositions"methodof computation.

3.25

```
1 A      »fO,  -8(»gp)
1 A      $f2,  -ie(tgp)
1A       Sf4,  -24(Sgp)
fmadd   tfO.  tfO,  tf2,  (f4
s.d      tfO,  -8($gp)
```

3.26 a.

1 = 0100 0000 0110 0000 0000 00000010 0001

y = 0100 0000 1010 0000 0000 0000 0000 0000

Exponents

```
  100 00000
 +100 0000 1
 -----------
 1000 0000 1
 -01111111
 -----------
  100 0001 0
```

```
X                          1.100 0000 0000 0000 0010 0001
y                         xl.010 0000 0000 0000 0000 0000
                          -------------------------------

 1  100 0000 0000 0000 0010 0001 000 0000 0000 0000 0000 0000
 +   11 0000 0000 0000 0000 1000 010 0000 0000 0000 0000 0000
-------------------------------------------------------------
 1.111 0000 0000 0000 0010 1001 010 0000 0000 0000 0000 0000
```

Round result for part b.

```
1.111  1100 0000 0000 0010 1001

Z0011  1100  111000000000 1010  11000000
```

Exponents

```
  100  0001  0
-  11  1100  1
--------------
        100  1   --> shift 9 bits

 1.1110000 0000 0000 0010  1001010 0000 00
+    z            1110000000000101011000000
------------------------------------------
1.111  OOOOOIUOOOOOOOIO  1110  101
                               GRS
```

Result:

```
0100  0001011100000111000000100  1111
```

b.

```
 1.111  1100 0000 0000 0000  1001   result from mult.
 +   z           1110000 0000 0101011
--------------------------------------
 1.111  11000111 0000 0001  1110011
                              GRS

0100000101110000  01110000  01001110
```

3.27

a.

```
                                                                      1 1     1 1 1
  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 1 0 1     1 0 1 1
+• 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0     1 1 0 1
 ---------------------------------------------------------------------------
  0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 1 1 0   1 0 0 0
```

b.

```
  0 0 0 0  0 0 0 0  0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 1 0 1   1 0 1 1
- 0 0 0 0   0 0 0 0  0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   1 1 0 1
                                                                     1   1
 ---------------------------------------------------------------------------
  0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 1 0 0   1 1 1 0
```

c.

```
  0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 1 0 1   1 0 1 1
x 0 0 0 o . 0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   1 1 0 1
 ---------------------------------------------------------------------------
                                                              1

                                                    1 1 1   1 1 1 1
  0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 '0 0 0   0 0 0 0   0 1 0 1   1 0 1 1
  0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 1   0 1 1 0   1 1
+ 0 0 0 0   o 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0   0 0 1 0   1 1 0 1   1
 ---------------------------------------------------------------------------
  n n n n   n n 0 0   0 n n n   n ▷ n 0   0 0 0 0   0 1 0 0   1 0 0 1   1 1 1 1
```

d.

```
                       0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 1 1 It™
                 ----------------------------------------------------------------------------
0 0 0 0  1 1 0 1 1 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 1 0 1  1 0 1 1
                     _ 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 1 1  0 1 | 1
                 ---------------------------------------------------------------------| |
                                                                          0 1 0   0 1 1 1
                     - 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 1  1 0 1 1
                 ----------------------------------------------------------------------|
                                                                              0   1 1 0 1
                     - 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  1 1 0 1
                 ----------------------------------------------------------------------------
                                                                              0 0 0 0
```

3.28

a.

```
                                                   1 1 1 1     1           1 1 1
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1  1 0 1 1  0 0 1 1  0 1 0 1  0 0 1 1
+ 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 1 0  1 1 0 1  0 1 1 1
  ---------------------------------------------------------------------
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1  1 0 1 1  0 1 1 0  0 0 1 0  1 0 1 0
```

b.

```
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1  1 0 1 1  0 0 1 1  0 1 0 1  0 0 1 1
- 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 1 0  1 1 0 1  0 1 1 1
                                                     1  1 1 1 1  1
  ---------------------------------------------------------------------
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1  1 0 1 1  0 0 0 0  0 1 1 1  1 1 0 0
```

c.

```
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1  1 0 1 1  0 0 1 1  0 1 0 1  0 0 1 1
× 0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0  0 0 1 0  1 1 0 1  0 1 1 1
  ---------------------------------------------------------------------
  6 6 6 6  6 6 6 6  6 5 5 5  4 4 4 4  3 4 3 3  4 3 3 2  1 1 1 1  1 1
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1  1 0 1 1  0 0 1 1  0 1 0 1  0 0 1 1
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 1  0 1 1 0  0 1 1 0  1 0 1 0  0 1 1
  1 1 1 1  1 1 1 1  1 1 1 1  1 1 1 0  1 1 0 0  1 1 0 1  0 1 0 0  1 1
  1 1 1 1  1 1 1 1  1 1 1 1  1 0 1 1  0 0 1 1  0 1 0 1  0 0 1 1
  1 1 1 1  1 1 1 1  1 1 1 0  1 1 0 0  1 1 0 1  0 1 0 0  1 1
  1 1 1 1  1 1 1 1  1 1 0 1  1 0 0 1  1 0 1 0  1 0 0 1  1
+ 1 1 1 1  1 1 1 1  0 1 1 0  0 1 1 0  1 0 1 0  0 1 1
  ---------------------------------------------------------------------
  1 1 1 1  1 1 1 1  0 0 1 0  0 1 1 0  0 1 0 0  0 0 0 0  1 0 1 1  0 1 0 1
```

d. Convert to positive:

```
                                                  1 1011
                   _____________________________________
0010 1101 011110000 0000  0000  0000 0100 1100 1010 1101
                                    0010 11010111  I   I
                          -  ___________________   |   |
                                     1 1111 0011 II    |
                                     1 0110 1011 II    |
                          -  ___________________   |   |
                                       1000 1000 0101
                                        101 1010 1111
                          -  _______________________|
                                         10 1101 0111
             -                           10 1101 0111
                               -  ______________________
                                         00 0000 0000
```

Since signs differ, convert quotient to negative;

1111 1111 1111 1111 1111 1111 11100101^

3.29 Start

Set subtract bit to true

1. If subtract bit true: Subtract the Divisor register from the Remainder and place the result in the remainder register.

   else Add the Divisor register to the Remainder and place the result in the remainder register.

   Test Remainder

   >=0

2. a. Shift the Quotient register to the left, setting rightmost bit to 1.

   <0

2. b. Set subtract bit to false.

3. Shift the Divisor register right 1 bit.

   <33rd rep —> repeat

   Test remainder

   <0

Add Divisor register to remainder and place in Remainder register.

Done

Example:

Perform n + 1 iterations for n bits

Remainder 0000 1011

Divisor 00110000

Iteration 1:

(subtract)

Rem 1101 1011

Quotient 0

Divisor 0001 1000

Iteration 2:

(add)

Rem 11110011

Q00

Divisor 0000 1100

Iteration 3:

(add)

Rem 11111111

Q0O0

Divisor 0000 0110

Iteration 4:

(add)

Rem 0000 0101

Q0001

Divisor 0000 0011

Iteration 5:

(subtract)

Rem 0000 0010

Q 0001 1

Divisor 0000 0001

Since remainder is positive, done.

Q = 0011 and Rem = 0010

3.30

a. -1 391 460 350

b. 2 903 506 946

c. -8.18545 XMT^{12}

d.sw $sO, »tO(16) sw Jrl6, Sr8(2)

3.31

a. 613 566 756

b. 613 566 756

c. 6.34413 X 10"^{17}

d.addiu. $s2, taO, 18724 addiu $18, 14, 0x8924

3.35

.285 X 10*
+9.84 X 10*

10.125 X10^4

1.0125 X 10*

with guard and round: 1.01 X 10^5

without: 1.01 X 10^5

3.36

3.63 X 10^4
+.687 X 10^4

4.317 X10^4

with guard and round: 4.32 x 10^4

without: 4.31 X 10^4

3.37

$$20_{ten} = 10100_{two} = 1.0100_{two}$$

Sign = 0, Significand – .01$\overline{0}$
Single exponent = 4 + 127 = 131
Double exponent = 4 + 1023 = 1027

Single precision	0 1000 Oil 010 0000 0000 0000 0000 0000
Double precision	0 1000 0000 011 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

3.38

Single precision	0 1000 0011 010 0100 0000 0000 0000 0000
Doub,e precision	I 0 1000 0000 Oil 0100 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 I

3.39

°-^{l}*un* = 0.0$\overline{0011}$^= U0$\overline{01L}$™* 2"*

Sign = 0, Significand = . $\overline{10011}$

Single exponent = -*A* + 127 = 123

Double exponent = -4 + 1023 = 1019

	← 1 →	← 8 →	← 23 →
Single precision	0	01111011	10011001100110011001100 trunc 10011001100110011001101 round

	← 1 →	← 11 →	← 52 →
Double precision	0	01111111011	lOOllOOllOOllOOllOOllOOllOOllOOllOOllOOllOOllOOllOOl trunc lOOllOOllOOllOOllOOllOOllOOllOOilOOllOOllOOllOOllOlO round

3.40

Single precision

```
I 1 00011110 101 01010101 0101 0101 0101
```

Double precision

```
1 0111 1111 110 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010   trunc
1 0111 1111 110 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1010 1011   round
```

3.41 No, since floating point adds are not associative, doing them simultaneously is not the same as doing them serially.

3.42

a.

Convert+1.1011 • 2^{11} + -1.11 * 2^{-2}

```
  1.1011 0000 0000 0000 0000 000
-0.0000 0000 0000 0001 1100 000
-------------------------------
  1.1010 1111 1111 11100100 000
```

oiooono noi oiii mi mi ooiooooo

b. Calculate new exponent:

```
  111111
  100 0110 1
+01111101
----------
1000 0101 0
-011 11111    minusbias
1111 1111
----------
 100 01011    new exponent
```

Multiply significands:

```
                                  1.101  1000 0000 0000 0000 0000
                                  xi.no oooo oooo oooo oooo oooo
  11111
      1  1011 0000 0000 0000 0000 0000 0000 0000 0000 0000 oooo
     11 0110 0000 0000 0000 0000 oooo 0000 0000 0000 oooo 0000
+1.10   1100 oooo 0000 0000 0000 0000 0000 0000 0000 0000 0000
--------------------------------------------------------------
10.11    1101 0000 00000000 0000 0000 0000 oooo 0000 0000 0000
```

Normalize and round:

exponent 100 0110 0
signifkand

1.011 10100000 0000 00000000

Signs differ, so result is negative:

1100 0110 0011 1010 0000 0000 0000 0000

3.43

0 101 1111 10111110 01000000 0000 0000
01010001 11111000 0000 0000 0000 0000

a. Determine difference in exponents:

```
  1011 1111
 -1010 0011
 ----------
   0011100--> 28
```

Add signiiicands after scaling:

```
  1.011 1110 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
 +0.000 0000 0000 0000 0000 0000 0000 1111 1000 0000 0000 0000 0000
 ------------------------------------------------------------------
  1.011 11100100 0000 00000000 0000 1111 1000 0000 0000 0000 0000
```

Round (truncate) and repack:

0 1011111 1011 1110 0100 000000000000.
0101 1111101111100100 0000 0000 0000

b. Trivially results in zero:

0000 0000 0000 0000 0000 0000 0000 0000

c. We are computing $(x + y) + z$, where $z = -x$ and $y * 0$
$(x + y) + -x = y$ intuitively
$(x + y) + -x = 0$ with finite floating-point accuracy

3.44

a. $2^{15} - 1 = 32767$

b.

2.1» «,ı $\times 2^{2^{15}}$

$2^{2^{11}} = 3.23 \text{ X } 10^{616}$

$2^{2^{12}} = 1.04\text{xl}0^{1233}$

$2^{2^{13}} = 1.09\text{xl}0^{24"}$

$2^{2"};= 1.19\text{xlO}^{4932}$

$2^{2^{15}} = 1.42 \text{ X } 10^{9,64}$

so

as small as 2.0_{wn} X $10^{"9864}$
and almost as large as 2.0_{ten} X 10^{9864}

c. 20% more significant digits, and 9556 orders of magnitude more flexibility. (Exponent is 32 times larger.)

3.45 The implied 1 is counted as one of the significand bits. So, 1 sign bit, 16 exponent bits, and 63 fraction bits.

3.46

Load 2 X $10^{!<"}$
Square it 4 x 10^{616}
Square it 1.6 x 10^{1233}
Square it 2.5 X 10^{2466}
Square it 6.2 X 10^{4932}
Square it 3.6 X 10^{986s}

Min 6 instructions to utilize the full exponent range.

Solutions for Chapter 4 Exercises

4.1 For PI, M2 is 4/3 (2 sec/1.5 sec) times as fast as Ml. For P2, Ml is 2 times as fast (10 sec/5 sec) as M2.

4.2 We know the number of instructions and the total time to execute the program. The execution rate for each machine is simply the ratio of the two values. Thus, the instructions per second for PI on Ml is (5 x 10^9 instructions/2 seconds) = 2.5 x 10^9 IPS, and the instructions for PI on M2 is (6 x 10^9 instructions/1.5 seconds) = 4x 10^9 IPS.

4.3 M2 runs 4/3 as fast as Ml, but it costs 8/5 as much. As 8/5 is more than 4/3, Ml has the better value.

4.6 Running PI 1600 times on Ml and M2 requires 3200 and 2400 seconds respectively. This leaves 400 seconds left for Ml and 1200 seconds left for M2. In that time Ml can run (400 seconds/{5 seconds/iteration)) = 80 iterations of P2. M2 can run (1200 seconds/(10 seconds/iteration)) = 120 iterations. Thus M2 performs better on this workload.

Looking at cost-effectiveness, we see it costs ($500/(80 iterations/hour)) = $6.25 per (iteration/hour) for Ml, while it costs ($800/(120 iterations/hour)) = $6.67 per (iteration/hour) for M2. Thus Ml is most cost-effective.

4.7

a. Time = (Seconds/cycle) * (Cycles/instruction) * (Number of instructions)

   Therefore the expected CPU time is (1 second/5 x 10^9 cycles) * (0.8 cycles/instruction) * (7.5 x 10^9 instructions) = 1.2 seconds

b. P received 1.2 seconds/3 seconds or 40% of the total CPU time.

4.8 The ideal instruction sequence for PI is one composed entirely of instructions from class A (which have CPI of 1). So Mi's peak performance is (4 x 10^9 cydes/second)/(1 cycle/instruction) = 4000 MIPS.

Similarly, the ideal sequence for M2 contains only instructions from A, B, and C (which all have a CPI of 2). So M2's peak performance is (6 x 10^9 cycles/second)/ (2 cycles/instruction) = 3000 MIPS.

4.9 The average CPI of PI is (1x2 + 2 + 3 + 4 + 3)/6 = 7/3. The average CPI of P2 is (2x2 + 2 + 2 + 4 + 4)/6 = 8/3. P2 then is ((6 x 10^9 cydes/second)/(8/3 cycles/instruction))/((4 x 10^9 cydes/second)/(7/3 cydes/instruction)) = 21/16 times faster than PI.

4.10 Using Cl, the average CPI for II is (.4 * 2 + .4 * 3 + .2 * 5) = 3, and the average CPI for 12 is (.4 * 1 + .2 * 2 + .4 * 2) = 1.6. Thus, with Cl, II is ((6 x 10^9 cycles/second)/^ cydes/instruction))/((3 x 10^9 cycles/second)/(1.6 cydes/instruction)) = 16/15 times as fast as 12.

Using C2, the average CPI for 12 is (.4 * 2 + .2 * 3 + .4 * 5) = 3.4, and the average CPI for 12 is (.4 * 1 + .4 * 2 + .2 * 2) = 1.6. So with C2,12 is faster than II by factor of ((3 x I0^9 cydes/second)/(1.6 cydes/instruction))/((6 x 10^9 cydes/second)/(3.4 cycles/instruction)) = 17/16.

For the rest of the questions, it will be necessary to have the CPIs of 11 and 12 on programs compiled by C3. For II, C3 produces programs with CPI (.6 * *2* + .15 * 3 + .25 * 5) = 2.9.12 has CPI (.6 * 1 + .15 * 2 + .25 * 2) = 1.4.

The best compiler for each machine is the one which produces programs with the lowest average CPI. Thus, if you purchased either II or 12, you would use C3.

Then performance of II in comparison to 12 using their optimal compiler (C3) is ({6 x 10^9 cydes/second)/(2.9 cydes/instmction))/((3 X 10^9 cydes/second)/(1.4 cycles/instruction)) = 28/29. Thus, 12 has better performance and is the one you should purchase.

4.11 Program P running on machine M takes (10^9 cydes/seconds) * 10 seconds = 10^{10} cydes. P^7 takes (10^9 cydes/seconds) * 9 seconds = 9 x 10^9 cydes. This leaves 10^9 less cycles after the optimization.

Everytime we replace a mult with two adds, it takes 4 - 2 * 1 = 2 cydes less per replacement.

Thus, there must have been 10^9 cydes /(2 cydes/replacement) = 5 X 10^8 replacements to make P into P'.

4.12 The first option reduces the number of instructions to 80%, but increases the time to 120%. Thus it will take: 0.8 * 1.2 = 0.96 as much time as the initial case.

The second option removes 20W2 = 10% of the instructions and increases the time taken to 110%. Therefore it will take 0.9 * 1.1 = 0.99 times as much time as the initial case.

Therefore, the first option is the faster of the two, and it is faster than the orginial, so you should have hardware automatically do garbage collection.

4.13 Let I = number of instructions in program and C = number of cydes in program. The six subsets are {dock rate, C} {cycle time, C} {MIPS, 1} {CPI, C, MIPS! {CPI, I, clock rate} {CPI, I, cyde time}. Note that in every case each subset has to have at least one rate {CPI, dock rate, cyde time, MIPSJ and one absolute {C, I}.

4.14 The total execution time for the machines are as follows:

Computer A =2 + 20 + 200 seconds = 222 seconds

Computer B =5 + 20 + 50 seconds = 75 seconds

Computer C = 10 + 20 + 15 seconds = 45 seconds

Thus computer C is fester. It is 75/45 = 5/3 times fester than computer B and 222/45 = 74/15 times faster than computer A.

4.15 With the new weighting the total execution time for the group of programs is:

Computer A = 8 * 2 + 2* 20 + 1 * 200 seconds = 256 seconds

Computer B = 8 * 5 + 2* 20 + 1 * 50 seconds = 130 seconds

Computer C = 8 * 10+ 2* 20 + 1 * 15 seconds = 135 seconds

So with this workload, computer B is faster by a factor of 135/130 = 1.04 with respect to computer C and a factor of 256/130 = 1.97 with respect to computer A. This new weighting reflects a bias from the previous results by a bias toward program 1 and program 2, which resulted in computer A and computer B looking comparitively better than before.

4.16 Comparing the times of the program executions on computer A, we see that to get an equal amount of execution time, we will have to run program 1100 times, program 2 10 times, and Program 3 1 time. This results in the following execution times:

Computer A = 100 * 2 + 10 * 20 + l * 200 seconds = 600 seconds

Computer B = 100 * 5 + 10 * 20 + 1 * 50 seconds = 750 seconds

Computer C = 100 * 10 + 10 * 20 + 1 * 15 seconds = 1215 seconds

So computer A is fastest with this workload.

Using computer B's program execution times to determine a weighting, we get a ratio of 20:5:2 for program 1, program 2, and program 3, respectively. This results in the following execution times:

Computer A = 20*2 + 5*20 + 2*200 seconds = 540 seconds

Computer B = 20!*5 + 5*20 + 2*50 seconds = 300 seconds

Computer C = 20 * 10 + 5 * 20 + 2 * 15 seconds = 330 seconds

So in this case, computer B is fastest.

Using the execution times on computer C, we get a 6:3:4 ratio, resulting in the following total execution times:

Computer A = 6 * 2 + 3* 20+ 4* 200 seconds = 872 seconds

Computer B = 6 * 5 + 3* 20+ 4* 50 seconds = 290 seconds

Computer C = 6 * 10+ 3* 20+ 4* 15 seconds = 180 seconds

So in this case computer C is fastest.

As we did the weighting to equalize the times on one particular machine, we ended up running programs that that computer could do the fastest most often and the programs that it was slower on less often. This will tend to be a comparative improvement in the execution time for the machine we are normalizing time to (as die weightings are not guaranteed to bias towards the programs that the other machines are better at). In this example, this type of weighting was enough to make that computer appear the fastest.

4.17 We know CPI is equal to (Cydes/second)/(Instructions/second). So the CPI of PI on Ml is (4 x 10^9 cydes/second)/(2.5 x 10* instructions/second) = 1.6 CPI, and the CPI of PI on M2 is (6 x 10^9 cydes/second)/(4 x 10^9 instructions/second) = 1.5 CPI.

4.18 We have the CPI, the dock rate, and the total execution time, and we're trying to find the total number of instructions. Using the following equation:

(Cydes/instruction)/(Cydes/second) * Instructions = (Execution time)

We find that there are (5 seconds) * (4 x 10^9 cydes/second)/(0.8 cydes/instruction) = 12.5 x 10^9 instructions in P2 .on Ml, and (10 seconds) * (6 x 10^9 cydes/second)/(1.5 CPI) = 40 X 10^9 instructions in P2 on M2.

4.19 No solution provided.

4.20 No solution provided.

4.21 No solution provided.

4.22 Using Amdahl's law (or just common sense), we can determine the following:

- Speedup if we improve only multiplication = 100/(30 + 50 + 20/4) = 100/85 = 1.18.
- Speedup if we only improve memory access = 100/(30 + 50/2 + 20)) = 100/75=1.33.
- Speedup if both improvements are made = 100/(30 + 50/2 + 20/4) = 100/60 = 1.67.

4.23 The problem is solved algebraically and results in the equation

100/(r+ (100-X- Y) + *X/4)* = 100/CX+ (100 -*X*- *Y)* + 172)

where *X* = multiplication percentage and *Y* = memory percentage. Solving, we get memory percentage = 1.5 x multiplication percentage. Many examples thus exist: for example, multiplication = 20%, memory = 30%, other = 50%, or multiplication = 30%, memory = 45%, other = 25%, and so on.

a. 9A Q *A* $\text{Speedup} = \frac{\text{Execution time before improvement}}{\text{Execution time after improvement}}$

Rewrite the execution time equation:

Execution time after improvement = $\frac{\text{Execution time affected by improvement}}{\text{Amount or improvement}}$ + ^ *a t i a n*,imeunaffected

$$= \frac{\text{Execution time affected + Amount of improvement x Execution time unaffected}}{\text{Amount of improvement}}$$

Rewrite execution time affected by improvement as execution time before improvement X *f*, where /is the fraction affected. Similarly execution time unaffected.

- $\frac{\text{Execution time before improvement}}{\text{Amount of improvement}}$ x *f* + ^^ ^ befere improv ementx(*i* _fl

= (j – + (1 - *f*) 1 x Execution time before improvement
lAmount of improvement

c ,
bpeedup = Execution time before improvement / [7^+ (1 - *f*) | x Execution time before improvement
V. Amount of improvement *)*

$$\text{Speedup} = \frac{1}{\left(\frac{f}{\text{Amount of improvement}} + (1-f)\right)}$$

The denominator has two terms: the fraction improved *(f)* divided by the amount of the improvement and the fraction unimproved (1 - /) .

4.25 We can just take the GM of the execution times and use the inverse.

GM(A) = VIOOO = 32, GM(B) = ,/l000 = 32,and GM(C) = ^400 = 20,

so C is fastest.

4.26 A, B: B has the same performance as A. If we run program 2 once, how many times should we run program 1? x + 1000 = IOJC + 100, or x = 100. So the mix is 99% program 1,1% program 2.

B, C: C is faster by the ratio of $\frac{32}{20}$ = 1.6.

Program 2 is run once, so we have 10JC+ 100= 1.6 x(20x+ 20), x= 3.1 times. So the mix is 76% program 1 and 24% program 2.

A, C: C is also faster by 1.6 here. We use the same equation, but with the proper times: *x*+ 1000= 1.6 x{20x+ 20), *x*= 31.2. So the mix is 97% program 1 and 3% program 2. Note that the mix is very different in each case!

4.27 No solution provided.

4.28 No solution provided.

4.29 No solution provided.

4.30

Program	Computer A	Computer B	Computer C
1	10	1	0.5
2	0.1	1	5

4.31 The harmonic mean of a set of rates,

$$\mathrm{HM} = \frac{n}{\sum_{i=1}^{n} \frac{1}{\mathrm{Rate}_i}} = \frac{n}{\sum_{i=1}^{n} \mathrm{Time}_i} = \frac{1}{\frac{\sum_{i=1}^{n} \mathrm{Time}_i}{n}} = \frac{1}{\frac{1}{n}\sum_{i=1}^{n} \mathrm{Time}_i} = \frac{1}{\mathrm{AM}}$$

where AM is the arithmetic mean of the corresponding execution times.

4.32 No solution provided.

4.33 The time of execution is (Number of instructions) * (CPI) * (Clock period). So the ratio of the times (the performance increase) is:

$$10.1 = \frac{\text{(Number of instructions) * (CPI) * (Clock period)}}{\text{(Number of instructions w/opt.) * (CPI w/opt.) * (Clock period)}}$$

$$= 1/\text{(Reduction in instruction count) * (2.5 improvement in CPI)}$$

Reduction in instruction count = .2475.

Thus the instruction count must have been reduced to 24.75% of the original.

4.34 We know that

(Number of instructions on V) * (CPI on V) * (Clock period)

$$5 = \frac{\text{(Time on V)}}{\text{(Time on P)}} = \frac{\text{(Number of instructions on V) * (CPI on V) * (Clock period)}}{\text{(Number of instructions on P) * (CPI on P) * (Clock period)}}$$

5 = (1/1.5) * (CPI of V)/(1.5 CPI)

CPI of V= 11.25.

4.45 The average CPI is .15 * 12 cycles/instruction + .85 * 4 cycles/instruction = 5.2 cycles/instructions, of which .15 * 12 = 1.8 cycles/instructions of that is due to multiplication instructions. This means that multiplications take up 1.8/5.2 = 34.6% of the CPU time.

4.46 Reducing the CPI of multiplication instructions results in a new average CPI of .15 * 8 + .85 * 4 = 4.6. The clock rate will reduce by a factor of 5/6 . So the new performance is (5.2/4.6) * (5/6) = 26/27.6 times as good as the original. So the modification is detrimental and should not be made.

4.47 No solution provided.

4.48 Benchmarking suites are only useful as long as they provide a good indicator of performance on a typical workload of a certain type. This can be made untrue if the typical workload changes. Additionally, it is possible that, given enough time, ways to optimize for benchmarks in the hardware or compiler may be found, which would reduce the meaningfulness of the benchmark results. In those cases changing the benchmarks is in order.

4.49 Let Tbe the number of seconds that the benchmark suite takes to run on Computer A. Then the benchmark takes 10 * T seconds to run on computer B. The new speed of A is (4/5 * *T*+ 1/5 * (T/50)) = 0.804 Tseconds. Then the performance improvement of the optimized benchmark suite on A over the benchmark suite on B is 10 * T/(0.804 *T)* = 12.4.

4.50 No solution provided.

4.51 No solution provided.

4.82 No solution provided.

Solutions for Chapter 5 Exercises

5.1 Combinational logic only: a, b, c, h, i

Sequential logic only: f, g, j

Mixed sequential and combinational: d, e, k

5.2

a. RegWrite = 0: All R-format instructions, in addition to 1 w, will not work because these instructions will not be able to write their results to the register file.

b. ALUopl = 0: All R-format instructions except subtract will not work correctly because the ALU will perform subtract instead of the required ALU operation.

c. ALUopO = 0: beq instruction will not work because the ALU will perform addition instead of subtraction (see Figure 5.12), so the branch outcome may be wrong.

d. Branch (or PCSrc) = 0: beq will not execute correctly. The branch instruction will always be not taken even when it should be taken.

e. MemRead = 0: 1 w will not execute correctly because it will not be able to read data from memory.

f. MemWrite = 0: sw will not work correctly because it will not be able to write to the data memory.

S.3

a. RegWrite = 1: sw and beq should not write results to the register file, sw (beq) will overwrite a random register with either the store address (branch target) or random data from the memory data read port.

b. ALUopO = 1: 1 w and sw will not work correctly because they will perform subtraction instead of the addition necessary for address calculation.

c. ALUopl = 1: 1 w and sw will not work correctly. 1 w and sw will perform a random operation depending on the least significant bits of the address field instead of addition operation necessary for address calculation.

d. Branch = 1: Instructions other than branches (beq) will not work correctly if the ALU Zero signal is raised. An R-format instruction that produces zero output will branch to a random address determined by its least significant 16 bits.

e. MemRead = 1: All instructions will work correctly. (Data memory is always read, but memory data is never written to the register file except in the case oflw.)

f. MemWrite = 1: Only sw will work correctly. The rest of instructions will store their results in the data memory, while they should not.

5.7 No solution provided.

5.8 A modification to the datapath is necessary to allow the new PC to come from a register (Read data 1 port), and a new signal (e.g., JumpReg) to control it through a multiplexor as shown in Figure 5.42.

A new line should be added to the truth table in Figure 5.18 on page 308 to implement the jr instruction and a new column to produce the JumpReg signal.

5.9 A modification to the data path is necessary (see Figure 5.43) to feed the shamt field (instruction [10:6]) to the ALU in order to determine the shift amount The instruction is in R-Format and is controlled according to the first line in Figure 5.18 on page 308.

The ALU will identify the s11 operation by the ALUop field.

Figure 5.13 on page 302 should be modified to recognize the opcode of si1; the third line should be changed to 1X1X0000 0010 (to discriminate the add and ss1 functions), and a new line, inserted, for example, 1X0X0000 0011 (to define si1 by the 0011 operation code).

5.10 Here one possible 1ui implementation is presented:

This implementation doesn't need a modification to the datapath. We can use the ALU to implement the shift operation. The shift operation can be like the one presented for Exercise 5.9, but will make the shift amount as a constant 16. A new line should be added to the truth table in Figure 5.18 on page 308 to define the new shift function to the function unit. (Remember two things: first, there is no funct field in this command; second, the shift operation is done to the immediate field, not the register input.)

RegDst = 1: To write the ALU output back to the destination register (trt).

ALUSrc = 1: Load the immediate field into the ALU.

MemtoReg = 0: Data source is the ALU.

RegWrite = 1: Write results back.

MemRead = 0: No memory read required.

MemWrite = 0: No memory write required.

Branch = 0: Not a branch.

ALUOp = 11: si1 operation.

This ALUOp (11) can be translated by the ALU asshl,ALUI1.16by modifying the truth table in Figure 5.13 in a way similar to Exercise 5.9.

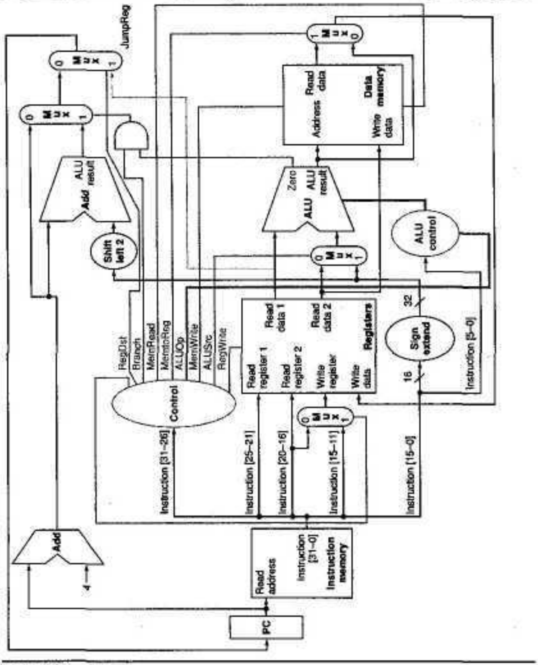

JumpReg
M u x
0
1
M u x
0
1
M u x
1
0
Add
ALU result
Shift left 2
Read data
Data memory
Address
Write data
Zero
ALU
ALU result
ALU control
M u x
0
1
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Read data 1
Read data 2
Registers
Read register 1
Read register 2
Write register
Write data
32
Sign extend
16
Instruction [5–0]
Control
M u x
0
1
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Add
4
Read address
Instruction [31–0]
Instruction memory
PC

FIGURE 5.42

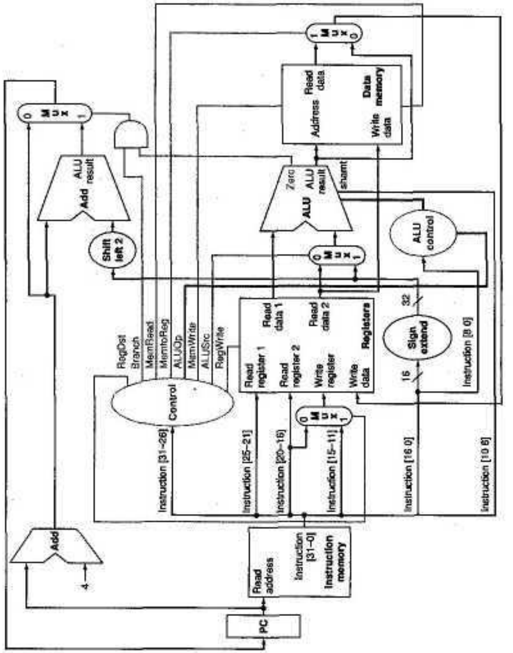

PC
Read address
Instruction [31–0]
Instruction memory
Add
4
Add
ALU result
Shift left 2
M u x
0
1
Control
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15 0]
Instruction [10 6]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
32
Sign extend
Instruction [8 0]
ALU control
Zero
ALU
ALU result
shamt
Address
Write data
Read data
Data memory

FIGURE 5.43

5.U A modification is required for the datapath of Figure 5.17 to perform the autoincrement by adding 4 to the $rs register through an incrementer. Also we need a second write port to the register file because two register writes are required for this instruction. The new write port will be controlled by a new signal, "Write 2", and a data port, "Write data 2." We assume that the Write register 2 identifier is always the same as Read register 1 {$rs). This way "Write 2" indicates that there is second write to register file to the register identified by "Read register 1," and the data is fed through Write data 2.

A new line should be added to the truth table in Figure 5.18 for the 1_inc command as follows:

RegDst = 0: First write to $rt.

ALUSrc = 1: Address field for address calculation.

MemtoReg = 1: Write loaded data from memory.

RegWrite = 1: Write loaded data into $rt.

MemRead = 1: Data memory read.

MemWrite = 0: No memory write required.

Branch = 0: Not a branch, output from the PCSrc controlled mux ignored.

ALUOp = 00: Address calculation.

Write2 = 1: Second register write (to $rs).

Such a modification of the register file architecture may not be required for a multiple-cycle implementation, since multiple writes to the same port can occur on different cycles.

5.12 This instruction requires two writes to the register file. The only way to implement it is to modify the register file to have two write ports instead of one.

5.13 From Figure 5.18, the MemtoReg control signal looks identical to both signals, except for the don't care entries which have different settings for the other signals. A don't care can be replaced by any signal; hence both signals can substitute for the MemtoReg signal.

Signals ALUSrc and MemRead differ in that sw sets ALSrc (for address calculation) and resets MemRead (writes memory: can't have a read and a write in the same cycle), so they can't replace each other. If a read and a write operation can take place in the same cycle, then ALUSrc can replace MemRead, and hence we can eliminate the two signals MemtoReg and MemRead from the control system.

Insight: MemtoReg directs the memory output into the register file; this happens only in loads. Because sw and beq don't produce output, they don't write to the register file (Regwrite = 0), and the setting of MemtoReg is hence a don't care. The important setting for a signal that replaces the MemtoReg signal is that it is set for 1 w (Mem->Reg), and reset for R-format (ALU->Reg), which is the case for the ALUSrc (different sources for ALU identify 1 w from R-format) and MemRead (1 w reads memory but not R-format).

5.14 swap $rs,$rt can be implemented by

```
addi $rd,$rs,0
addi $rs,$rt,0
addi $rt,$rd,0
```

if there is an available register $rd

or

```
sw $rs,temp($rO)
addi $rs,$rt,0
Iw $rt,temp($rO)
```

if not.

Software takes three cycles, and hardware takes one cycle. Assume *Rs* is the ratio of swaps in the code mix and that the base CPI is 1:

Average MIPS time per instruction = $Rs * 3 * T + (1 - Rs) * 1 * T = \{2Rs + 1) * T$

Complex implementation time = $1.1 * T$

If swap instructions are greater than 5% of the instruction mix, then a hardware implementation would be preferable.

. 5.27 l_incr $rt,Address(Irs) can be implemented as

```
?w trt.Address(trs)
addi $rs,$rs,l
```

Two cycles instead of one. This time the hardware implementation is more efficient if the load with increment instruction constitute more than 10% of the instruction mix.

5.28 Load instructions are on the critical path that includes the following functional units: instruction memory, register file read, ALU, data memory, and register file write. Increasing the delay of any of these units will increase the clock period of this datapath. The units that are outside this critical path are the two adders used for PC calculation (PC + 4 and PC + Immediate field), which produce the branch outcome.

Based on the numbers given on page 315, the sum of the the two adder's delay can tolerate delays up to 400 more ps.

Any reduction in the critical path components will lead to a reduction in the dock period.

5.29

a. RegWrite = 0: All R-format instructions, in addition to 1 w, will not work because these instructions will not be able to write their results to the register file.

b. MemRead = 0: None of the instructions will run correctly because instructions will not be fetched from memory.

c. MemWrite = 0: s w will not work correctly because it will not be able to write to the data memory.

d. IRWrite = 0: None of the instructions will run correctly because instructions fetched from memory are not properly stored in the IR register.

e. PCWrite = 0: Jump instructions will not work correctly because their target address will not be stored in the PC.

f. PCWriteCond = 0: Taken branches will not execute correctly because their target address will not be written into the PC.

5.30

a. RegWrite = 1: Jump and branch will write their target address into the register file, sw will write the destination address or a random value into the register file.

b. MemRead = 1: All instructions will work correctly. Memory will be read all the time, but IRWrite and IorD will safeguard this signal.

c. MemWrite = 1: All instructions will not work correctly. Both instruction and data memories will be written over by the contents of register B.

d. IRWrite= 1: lw will not work correctly because data memory output will be translated as instructions.

e. PCWrite = 1: All instructions except jump will not work correctly. This signal should be raised only at the time the new PC address is ready (PC + 4 at cycle 1 and jump target in cycle 3). Raising this signal all the time will corrupt the PC by either ALU results of R-format, memory address of 1 w/sw, or target address of conditional branch, even when they should not be taken.

f. PCWriteCond = 1: Instructions other than branches (beq) will not work correctly if they raise the ALU's Zero signal. An R-format instruction that produces zero output will branch to a random address determined by .their least significant 16 bits.

5.31 RegDst can be replaced by $\overline{\text{ALUSrc}}$, $\overline{\text{MemtoReg}}$, $\overline{\text{MemRead}}$, ALUopl.

MemtoReg can be replaced by $\overline{\text{RegDst}}$, ALUSrc, MemRead, or $\overline{\text{ALUOpl}}$.

Branch and ALUOpO can replace each other.

5.32 We use the same datapath, so the immediate field shift will be done inside theALU.

1. Instruction fetch step: This is the same (IR <= Memory[PCl; PC <= PC +• 4)

2. Instruction decode step: We don't really need to read any register in this stage if we know that the instruction in hand is a 1 u 1, but we will not know this before the end of this cycle. It is tempting to read the immediate field into the ALU to start shifting next cycle, but we don't yet know what the instruction is. So we have to perform the same way as the standard machine does.

A <= 0 ($rO); B <= $rt; ALUOut <= PC + (sign-extend(immediate field));

3. Execution: Only now we know that we have a 1 u i. We have to use the ALU to shift left the low-order 16 bits of input 2 of the multiplexor. (The sign extension is useless, and sign bits will be flushed out during the shift process.)

ALUOut <= {IR[15-OJ,16(O)J

4. Instruction completion: Reg[IR[20-16]] = ALUOut.

The first two cycles are identical to the FSM of Figure 5.38. By the end of the second cycle the FSM will recognize the opcode. We add the Op='lui', a new transition condition from state 1 to a new state 10. In this state we perform the left shifting of the immediate field: ALUSrcA = x, ALUSrcB = 10, ALUOp = 11 (assume this means left shift of ALUSrcB). State 10 corresponds to cycle 3. Cycle 4 will be translated into a new state 11, in which RegDst = 0, RegWrite, MemtoReg = 0. State 1*1* will make the transition back to state 0 after completion.

As shown above the instruction execution takes 4 cycles.

5.33 This solution can be done by modifying the data path to extract and shift the immediate field outside the ALU. Once we recognize the instruction as 1 u i (in cycle 2), we will be ready to store the immediate field into the register file the next cycle. This way the instruction takes 3 cycles instead of the 4 cycles of Exercise 5.26.

1. Instruction fetch step: Unchanged.
2. Instruction decode: Also unchanged, but the immediate field extraction and shifting will be done in this cycle as well.

3. Now the final form of the immediate value is ready to be loaded into the register file. The MemtoReg control signal has to be modified in order to allow its multiplexor to select the immediate upper field as the write data source. We can assume that this signal becomes a 2-bit control signal, and that the value 2 will select the immediate upper field.

Figure 5.44 plots the modified datapath.

The first two cycles are identical to the FSM of Figure 5.38. By the end of the second cycle, the FSM will recognize the opcode. We add the Op = 'lui', a new transition condition from state 1 to a new state 10. In this state we store the immediate upper field into the register file by these signals: RedDst = 0, RegWrite, MemtoReg = 2. State 10 will make the transition back to state 0 after its completion.

5.34 We can use the same datapath.

1. Instruction fetch: Unchanged (IR <= Memory[PC]; PC<= PC + 4).
2. Instruction decode: Unchanged (A <= Reg[IR[25-21]]; B<=REG[IR[20-16]];ALUOut<=PC+(sign-extend(IR[15-03)<<2).
3. Load immediate value from memory (MDR <= Memory[PC]; PC <= PC + 4).
4. Complete instruction (Reg[IR[20-16]] (dependent on instruction format) <= MDR).

The first two cycles are identical to the FSM of Figure 5.38.

We add the Op='ldi', a new transition condition from state 1 to a new state 10. In this state we fetch the immediate value from memory into the MDR: MemRead, ALUSrcA = 0, IorD = 0, MDWrite, ALUSrcB = 01, ALUOp = 00, PCWrite, PCSource = 00.

FSM then makes the transition to another new state 11.

In this state we store the MDR into the register file by these signals: RedDst = 0 (actually depends on the instruction format), RegWrite, MemtoReg = 1.

State 11 will make the transition back to state 0 after its completion.

Four cycles to complete this instruction, in which we have two instruction memory accesses.

5.35 Many solutions are possible. In all of them, a multiplexor will be needed as well as a new control signal (e.g., RegRead) to select which register is going to be read (i.e., using IR[25-ll]orIR[20-16]). One simple solution is simply to add a write signal to A and break up state 1 into two states, in which A and B are read. It is possible to avoid adding the write signal to A if B is read first. Then A is

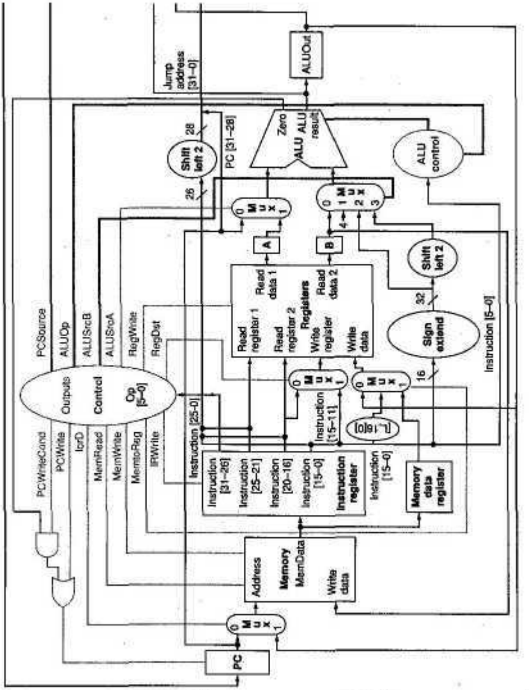

PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op [5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
Instruction [25–0]
Jump address [31–0]
26
Shift left 2
28
PC [31–28]
PC
0 M u x 1
Address
Memory
MemData
Write data
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–0]
Instruction register
Instruction [15–0]
Memory data register
Instruction [15–11]
[L,16]0
0 M u x 1
0 M u x 1
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
A
B
0 M u x 1
4
0 1 M u x 2 3
16
Sign extend
32
Shift left 2
Instruction [5–0]
ALU control
Zero
ALU
ALU result
ALUOut

FIGURE 5.44

read and RegRead is held stable (because A always writes). Alternatively, you could decide to read A first because it may be needed to calculate an address. You could then postpone reading B until state 2 and avoid adding an extra cycle for the load and store instructions. An extra cycle would be needed for the branch and R-type instructions.

6.36 Effective CPI = Sum(operation frequency * operation latency)

MIPS = Frequency/CPIeffective

Instruction	Frequency	Ml	M2	M3
Loads CPI	25%	5	4	3
Stores CPI	13%	4	4	3
R-type CPI	47%	4	3	3
Branch/jmp CPI	15%	3	3	3
Effective CPI		4.1	3.38	3
MIPS		976	946	933

From the results above, the penalty imposed on frequency (for all instructions) exceeds the gains attained through the CPI reduction. Ml is the fastest machine.

The more the load instructions in the instruction mix, the more the CPI gain we can get for the M2 and M3 machines. In the extreme case we have all instructions loads, Ml MIPS = 800, M2 MIPS = 300, and M3 MIPS = 933.3, so M3 becomes the best machine in such a case.

5.37 Effective CPI = Sum(operation frequency * operation latency)

MIPS = Frequency/CPIeffective

Instruction	Frequency	2.8 GHZ CPI	5.6 GHz CPI	6.4 GH CPI
Loads CPI	26%	5	6	7
Stores CPI	10%	4	5	6
R-type CPI	49%	4	4	5
Branch/jmp CPI	15%	3	3	4
Effective CPI		4.11	4.47	5.47
MIPS		1167.9	1250	1170.0

The two-cycle implementation increases the frequency, which benefits all instructions, and penalizes only loads and stores. The performance improvement is 7% relative to die original implementation.

Further increase of the clock frequency by increasing the instruction fetch time into two cycles will penalize all instructions and will reduce the performance to about the same as that of the 4.8 GHz base performance. Such implementation hurts the CPI more than the gain it brings through frequency increase and should not be implemented.

5.38

```
sit   $t4, $zero, $t3
      beg  $t4. $zero, exit
cmpr: lw   $t4, 0{$tl)
      lw   $t5, 0{$t5)
      bne  $t4, $t5, done
      addT $t3. $t3, -1
      addi ni,  $tl, 4
      addi" StZ, $t2, 4
      bne  $t3, Szero, cmpr
exit  addi $tl. Szero, $zero
done:
```

To compare two 100-work blocks we'll perform at most one sit 200 loads, 300 adds, and 201 branches = 803 instructions (if the two blocks are equal). Using this chapter's multicycle implementation, this will take 4 cycles for sit 1000 cycles for loads, 1200 cycles for adds, and 603 cycles for branches. The total cycles = 2811 cycles.

5.39 No solution provided.

5.49 No solution provided.

5.50 The exception cause can be represented through the status "cause" register, which records the reason code for the exception. The instruction position at which the exception occur is identified by saving it in the Exception Program Counter (EPC) register.

Execution can be restarted for some exceptions like overflow, system call request, or external I/O device interrupt by restarting execution at the EPC after handling the exception.

Other exceptions are not restartable and program has to terminate. Examples of this are invalid instructions (which can actually be restartable if defined as NOP by the hardware), power/hardware failure, and divide by zero. In such a case, an error message can be produced, and program termination takes place.

5.51

a. Divide by zero exception can be detected in the ALU in cycle 3, before executing the divide instruction.

b. Overflow can be hardware detected after the completion of the ALU operation. This is done in cycle 4 (see Figure 5.40)

c. Invalid opcode can be detected by the end of cycle 2 (see Figure 5.40).

d. This is an asynchronous exception event that can occur at any cycle. We can design this machine to test for this condition either at a specific cycle (and then the exception can take place only in a specific stage), or check in every cycle (and then this exception can occur at any processor stage).

e. Check for instruction memory address can be done at the time we update the PC. This can be done in cycle 1.

f. Check for data memory address can be done after address calculation at the end of cycle 3.

S.53 No solution provided.

5.57 No solution provided.

5.58 a) will assign the same value (2) to both A and B.

b) will swap A and B (A = 2 and B = 1).

5.59

```
module ALUControl (ALUOp, FuncCode, ALUCtl):
       input ALUOp[l:O], FuncCode[5:0];
                  output ALUCtl[3:0];
                     if(ALUOp — Z'b 00)
                        ALUCtl - 4'b 0010;
                     1f(ALUOp — Z'b 01)
                        ALUCtl = 4'b 0110;
                     iffALUOp — 2'b 10) begin
                        case(FuncCode)
                          6' b 100000: ALUCtl - 4'b 0010;
                          6'b 100O10: ALUCtl - 4'b 0110;
                          6'b 100100: ALUCtl - 4'b 0000;
                          6'b 100101: ALUCtl - 4'b 0001;
```

```
                6'b 101010: ALUCtl = 4'b 0111;
                default     ALUCtl - 4'b xxxx;
              endcase
           end
       endmodule
```

S.60

```
// Register File
module RegisterFile (Readl.Read2,Writereg,Writedata.Regwrite,
0atalData2,clock);
 input [5:0] Readl,Read2.Wn"tereg;  // the registers numbers to read
or write
 input [31:0] Writedata;      // data to write
 input RegUrite.              // The write control
       clock;                 // the clock to trigger writes
 output [31:0] Datal, Data2; // the register values read;
 reg [31:0] RF [31:0];        // 32 registers each 32 bits long
 initial RF[O] = 32"h 00000000; // Initialize all registers to 0
 always begin
   Datal <= RFCReadl]; Data2 <= RF[Read2];
   // write the register with new value if RegwMte is high
   @(negedge clock) if RegWrite then RF[Writereg] <= WriteData;
 end
endmodule
```

```
//ALU Control same as 5.30
module ALUControl (ALUOp, FuncCode. ALUCtl);
input ALUOp[l:0L FuncCode[5:0];
output ALUCtl[3:0];
   iffALUOp — 2'b 00)
      ALUCtl = 4'b 0010;
   if(ALU0p — 2'b 01)
      ALUCtl - 4'b 0110:
   1f(ALUOp = 2'b 10) begin
      case(funct)
         6'b 100000: ALUCtl = 4'b 0010;
         6'b 100010: ALUCtl - 4'b 0110:
         6'b 100100: ALUCtl = 4'b 0000;
         6'b 100101: ALUCtl = 4'b 0001;
         6'b 101010: ALUCtl = 4'b 0111;
         // .... Add more ALU control here
```

```
        default     ALUCtl = 4'b xxxx; //can report an error, or debug
information
      endcase
   end
endmoduie

//ALU
module HIPSALU (ALUctl. A, B, ALUOut, Zero):
input [3:0] ALUctl;
input [31:0] A,B;
output [31:0] ALUOut;
output Zero;
assign Zero " tALUOut—0); //Zero is true if ALUOut is 0
always @(ALUctl, A, B) begin //reevaluate if these change
  case (Aluctl)
    0: ALUOijt <- A & B;
    1: ALUOijt <- A 1 B;
    2: ALUOut <- A + B;
    6: ALUOiJt <-. A - B;
    7: ALUOijt <- A < B ? 1:0;
    // .... Add tnare ALU operations here
   default: ALUOut <= X;   //can report an error, or debug information
  endcase
 end
endmodule

//2-to-l Multiplexor
module Mult2tol Unl.In2.Sel.Out);
input [31:0] Inl, InZ;
input Sel;
output [31:0] Out:
always @(Inl, In2. Sel)
  case (Sel) //a 2->l multiplexor
    0: Out <= Inl;
    default: Out <- InZ;
  endcase;
endmodule;
```

```
//This represents every thing in Figure 5.19 on page 309 except the
"control block"
//Wnicn decodes the opcode, and generate the control signals
accordingly

module OataPathistart,RegDst.Branch,MemRead.HemtoReg.ALUOp,Mem-
Write,ALUSrc.RegWrite,opcode,clock}
input [31:0] start;      //PC initial value

input  RegDst.Branch,MemRead,MemtoReg,
       ALUOp,MemWrite.ALUSrc.RegWrite,clock;
input [1:0] ALUOp;
output [5:0] opcode;
initial begin          //initialize PC and Memories
    PC = start;
    IMemory = PROGRAM;
    DMemory - OATA;
end
reg [31:0] PC, IMemory[0:1023]. 0Memory[0:1023];
wire [31:0] SignExtendOffset, PCOffset, PCValue, ALUResultOut,
    IAddress. DAddress, IMemOut, DmemOut, DWriteData, Instruction,
    RWriteData. DreadData. ALUAin. ALUBin;
wire [3:0] ALUctl:
wi re Zero;
wire [5:0] WriteReg;  .
//Instruction fields, to improve code readability
wire [5:0]  funct;
wire [4:0]  rs, rt, rd. shamt;
wire [15:0] offset;
ALUControl alucontroller(ALUOp,Instruction[5:0],ALUctl);
    //ALL control
'MIPSALU ALUCALUctl, ALUAin, ALUBin, ALUResultOut, Zero);
RegisterFile REGtrs. rt, WriteReg,  RWriteOata, ALUAin, DWriteData'
clock);
Mult2tol regdst (rt, rd, RegDst, RegWrite),
     alusrc (DWriteData, SignExtendOffset. ALUSrc, ALUBin),
     pesre  (PC+4. PC+4+PC0ffset, Branch S Zero. PCValue);
assign {opcode, rs, rt, rd, shamt, funct} - Instruction;
assign offset = Instruction[15:0];
assign SignExtendOffset = I16{offset[15]},offset}; //sign~extend
lower 16 bits:
assign PCOffset = SignExtendOffset « 2;
```

```
always @(negedge clock) begin
  Instruction - IMemory[PC];
  PC <- PCValue;
 end
 always ©(posedge clock) begin
  if MemRead
     DreadData <- DMemoryLALUResultOut]:
  else 1f MemWrite
     DMemoryCALUResultOut] <- OWriteData;
  end
 end

endmodule

module MIPSlCYCLE(start);
// Clock
reg clock: // clock is a regfster
Initial clock - 0:
parameter LW - 6b 100011. SW - 6b 101011, BE0-6b 000100;
input start;
wire [1:0] AlUOp;
wire [5:0] opcode;
wire [31:0] SignExtend;
wire  RegDst,Branch.MemRead.MemtoReg.ALUOp.MemWrite.ALUSrc.RegWrite;
Datapath MIPSDP (start.RegDst.Branch,MemRead,MemtoReg.ALUOp,
MemWrite.ALUSrc.RegWrite.opcode.clock);
  //datapath control
always begin
 #1 clock =- - clock;  //clock generation
  case(opcode)
    0:     |RegDst,ALUSrc.MemtoReg.RegWrite,MemRead.MemWrite,Branch,
ALUOp}- 9'b 100100010;//R-Format
    LW:    IRegDst.ALUSrc.MemtoReg.RegWrite.MemRead,MemWrite,Branch,
ALUOp!- 9'b 011110000;
    SW:   ) RegDst, ALUSrcMemtoReg.RegWrite.MettiRead,MemWrite, Branch.
ALUOp)" 9'b xlxOOlOOO;
    BEQ:  (RegDst,ALUSrc.MemtoReg.RegWrite,MemRead,Mem-
Write. Branch.ALUOpH 9"b xOxOOOlOl;
    // .... Add more fnstructions here
    default: Sfinish;     // end simulation if invalid opcode
  endcase
end
endmodule
```

5.61 We implement the add shift functionality to the ALU using the Verilog code provided in B.22 in Appendix B. The 32-bit multiply execution takes 32 cycles to complete, so the instruction takes a total of 35 cycles. Assume the ALU control recognizes the multiply code correctly.

We follow the CD Verilog code, but we add the following:

```
case(state)
.
.
3: begin          //Execution starts at cycle 3
  state=4
  .
  .
  casefopcode—6'b 0)
    .
    .
    MPYU: begin
// issue load command to the multiplier
   !RegOst,ALUSrc,MemtoReg,RegWrite,MemRead,
MemWrite.Branch,ALUOpJ- 9"b 1001000110;//R-Format  same
command. Al u should now recognize the Func Field
    end
.
.
35:          // After 3? cycles the multiplication
results are available in the 32-bit Product output of
the ALU. Write the high order and low order words in
this and the next cycle.
  case(opcode—6'b 0) case <IR[5:0])
    .
    .
    MPYU: begin
  Regs[hi]=RegH
    end
```

```
34:
  case<opcode-=6'b 0) case (IR[5:0])
      .
      .
    MPYU: begin
  Regs[lo]-RegL
    end
end
```

5.62 We add the divide functionality to the ALU using the code of B.23. The rest of the solution is almost exactly the same as the answer to Exercise 5.61.

5.63 No solution provided

5.64 No solution provided.

5.65 No solution provided.

5.66 No solution provided.

Solutions for Chapter 6 Exercises

6.1

a. Shortening the ALU operation will not affect the speedup obtained from pipelining. It would not affect the dock cycle.

b. If the ALU operation takes 25% more time, it becomes the bottleneck in the pipeline. The clock cycle needs to be 250 ps. The speedup would be 20% less.

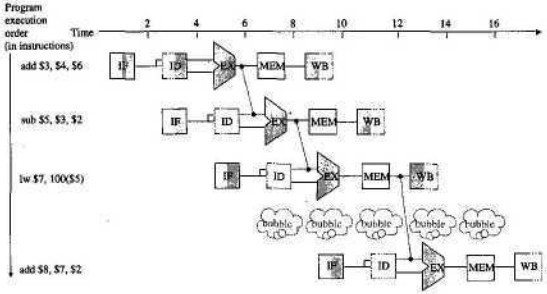
6.2

a. It takes 100 ps * 10^6 instructions - 100 microseconds to execute on a non-pipelined processor (ignoring start and end transients in the pipeline).

b. A perfect 20-stage pipeline would speed up the execution by 20 times.

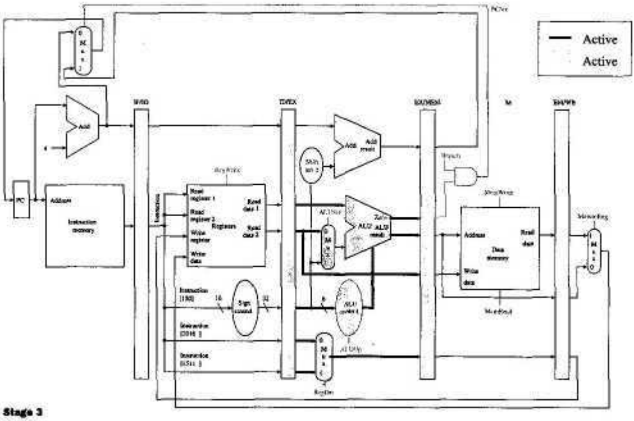
c. Pipeline overhead impacts both latency and throughput.

6.3 See the following figure:



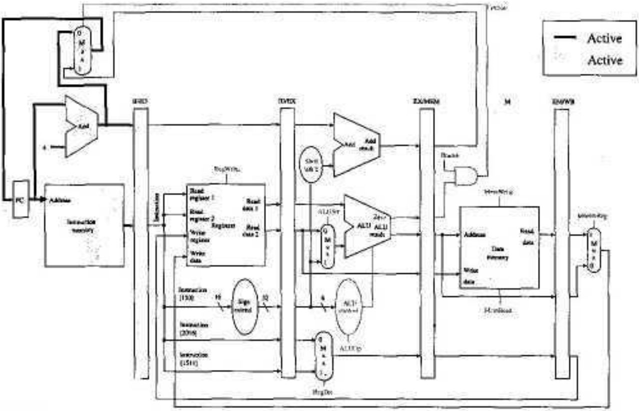
6.4 There is a data dependency through $3 between the first instruction and each subsequent instruction. There is a data dependency through $6 between the 1 w instruction and the last instruction. For a five-stage pipeline as shown in Figure 6.7, the data dependencies between the first instruction and each subsequent instruction can be resolved by using forwarding.

The data dependency between the load and the last add instruction cannot be resolved by using forwarding.

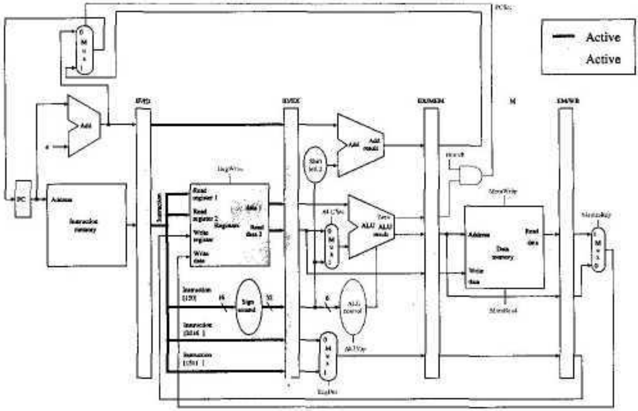

PCSrc
Active
Active
IF/ID
ID/EX
EX/MEM
MEM/WB
M
u
x
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
Branch
ALUSrc
Zero
ALU
ALU result
MemWrite
Address
Read data
Data memory
Write data
MemRead
MemtoReg
Instruction [15-0]
16
Sign extend
32
6
ALU control
ALUOp
Instruction [20-16]
Instruction [15-11]
RegDst

Stage 3

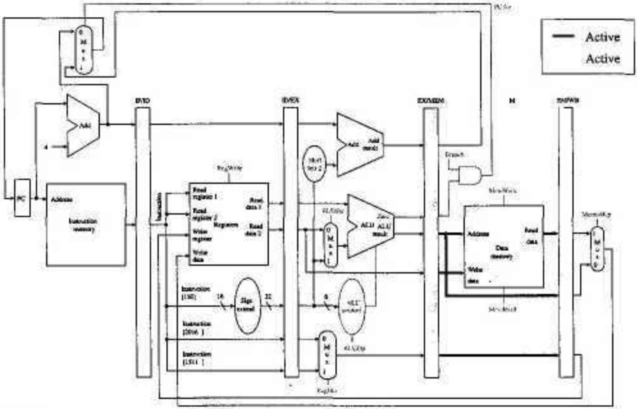
6.6 Any part of the following figure not marked as active is inactive.



Stage 1


Active
Active
PCSrc
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
Branch
ALUSrc
Zero
ALU
ALU result
MemWrite
Address
Data memory
Write data
Read data
MemRead
MemtoReg
Instruction [15-0]
16
Sign extend
32
6
ALU control
ALUOp
Instruction [20-16]
Instruction [15-11]
RegDst

Stage 2



Statf.4

Since this is an sw instruction, there is no work done in the WB stage.

6.12 No solution provided.

6.13 No solution provided.

6.14 No solution provided.

6.17 At the end of the first cycle, instruction 1 is fetched.

At the end of the second cycle, instruction 1 reads registers.

At the end of the third cycle, instruction 2 reads registers.

At the end of the fourth cycle, instruction 3 reads registers.

At the end of the fifth cycle, instruction 4 reads registers, and instruction 1 writes registers.

Therefore, at the end of the fifth cycle of execution, registers 16 and $ 1 are being read and register *$2* will be written.

6.18 The forwarding unit is seeing if it needs to forward. It is looking at the instructions in the fourth and fifth stages and checking to see whether they intend to write to the register file and whether the register written is being used as an ALU input. Thus, it is comparing 3 = 4? 3 = 2? 7 = 4? 7 = 2?

6.19 The hazard detection unit is checking to see whether the instruction in the ALU stage is an 1 w instruction and whether the instruction in the ID stage is reading the register that the 1 w will be writing. If it is, it needs to stall. If there is an 1 w instruction, it checks to see whether the destination is register 6 or 1 (the registers being read).

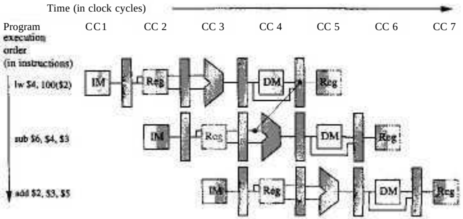
6.21

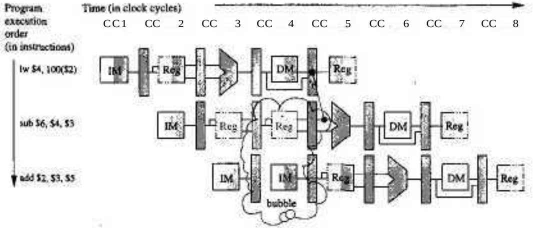
a. There will be a bubble of 1 cycle between a 1 w and the dependent add since the load value is available after the MEM stage.

   There is no bubble between an add and the dependent 1 w since the add result is available after the EX stage and it can be forwarded to the EX stage for the dependent 1 w. Therefore, CPI = cycle/instruction = 1.5.

b. Without forwarding, the value being written into a register can only be read in the same cycle. As a result, there will be a bubble of 2 cycles between an 1 w and the dependent add since the load value is written to the register after the MEM stage. Similarly, there will be a bubble of 2 cycles between an add and the dependent 1 w. Therefore, CPI = 3.

6.22 It will take 8 cycles to execute this code, including a bubble of 1 cycle due to the dependency between the 1 w and sub instructions.





6.23

Input	Number of bits	Usage
ID/£X.ReglsterRs	5	operand reg number, compare to see if match
ID/EX.RegisterRt	5	operand reg number, compare to see if match
EX/MEM.RegisterRd	5	destination reg number, compare to see if match
EX/MEM.RegWrite	1	TRUE if writes to the destination reg
MEM/WB.RegisterBd	5	destination reg number, compare to see if match
MEM/WB.RegWrlte	1	TRUE If writes to the destination reg

Output	Number of bits	Usage
ForwardA	2	forwarding signal
ForwardB	2	forwarding signal

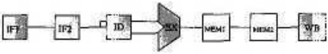
6.29 No solution provided.

6.30 The performance for the single-cycle design will not change since the clock cycle remains the same.

For the multicycle design, the number of cycles for each instruction class becomes the following: loads: 7, stores: 6, ALU instructions: 5, branches: 4, jumps: 4.

CPI = 0.25 * 7 + 0.10 * 6 + 0.52 * 5 + 0.11 * 4 + 0.02 * 4 = 5.47. The cycle time for the multicycle design is now 100 ps. The average instruction becomes 5.47 * 100 = 547 ps. Now the multicycle design performs better than the single-cycle design.

6.33 See the following figure.



when defined by lw	when defined by R-type
used in i1 => 2-cycle stall	used in i1 => forward
used In i2 => 1-cycle stall	used lin\2 => forward
used in i3 => forward	used ii i3 => forward

6.34 Branches take 1 cycle when predicted correctly, 3 cycles when not (including one more memory access cycle). So the average dock cycle per branch is 0.75 * 1 + 0.25 * 3 = 1.5.

For loads, if the instruction immediately following it is dependent on the load, the load takes 3 cycles. If the next instruction is not dependent on the load but the second following instruction is dependent on the load, the load takes two cycles. If neither two following instructions are dependent on the load, the load takes one cycle.

The probability that the next instruction is dependent on the load is 0.5. The probability that the next instruction is not dependent on the load, but the second following instruction is dependent, is 0.5 * 0.25 = 0.125. The probability that neither of the two following instructions is dependent on the load is 0.375.

Thus the effective CPI for loads is 0.5 * 3 + 0.125 * 2 + 0.375 * 1 = 2.125.

Using the date from the example on page 425, the average CPI is 0.25 * 2.125 + 0.10 * 1 + 0.52 * 1 + 0.11 * 1.5 + 0.02 * 3 = 1.47.

Average instruction time is 1.47 * lOOps = 147 ps. The relative performance of the restructured pipeline to the single-cycle design is 600/147 = 4.08.

6.35 The opportunity for both forwarding and hazards that cannot be resolved by forwarding exists when a branch is dependent on one or more results that are still in the pipeline. Following is an example:

```
lw  $1. $2(100)
add $1, $1. 1
beq $1, $2, 1
```

6.36 Prediction accuracy = 100% * PredictRight/TotalBranches

a. Branch 1: prediction: T-T-T, right: 3, wrong: 0

Branch 2: prediction: T-T-T-T, right: 0, wrong: 4

Branch 3: prediction: T-T-T-T-T-T, right: 3, wrong: 3

Branch 4: prediction: T-T-T-T-T, right: 4, wrong: 1

Branch 5: prediction: T-T-T-T-T-T-T, right: 5, wrong: 2

Total: right: 15, wrong: 10

Accuracy = 100% * 15/25 = 60%

b. Branch 1: prediction: N-N-N, right: 0, wrong: 3

Branch 2: prediction: N-N-N-N, right: 4, wrong: 0

Branch 3: prediction: N-N-N-N-N-N, right: 3, wrong: 3

Branch 4: prediction: N-N-N-N-N, right: 1, wrong: 4

Branch 5: prediction: N-N-N-N-N-N-N, right: 2, wrong: 5

Total: right: 10, wrong: 15

Accuracy - 100% * 10/25 - 40%

c. Branch 1: prediction: T-T-T, right: 3, wrong: 0

Branch 2: prediction: T-N-N-N, right: 3, wrong: 1

Branch 3: prediction: T-T-N-T-N-T, right: 1, wrong: 5

Branch 4: prediction: T-T-T-T-N, right: 3, wrong: 2

Branch 5: prediction: T-T-T-N-T-T-N, right: 3, wrong: 4

Total: right: 13, wrong: 12

Accuracy = 100% * 13/25 = 52%

d. Branch 1: prediction: T-T-T, right: 3, wrong: 0

Branch 2: prediction: T-N-N-N, right: 3, wrong: 1

Branch 3: prediction: T-T-T-T-T-T, right: 3, wrong: 3

Branch 4: prediction: T-T-T-T-T, right: 4, wrong: 1

Branch 5: prediction: T-T-T-T-T-T-T, right: 5, wrong: 2

Total: right: 18, wrong: 7

Accuracy = 100% * 18/25 = 72%

6.37 No solution provided.

6.38 No solution provided.

6.39 Rearrange the instruction sequence such that the instruction reading a value produced by a load instruction is right after the load. In this way, there will be a stall after the load since the load value is not available till after its MEM stage.

```
lw  $2. 100($6)
add $4. $2, $3
lw  $3, 200($7)
add $6, $3, $5
sub $8, 14, $6
lw  $7, 300($8)
beq $7, 18, Loop
```

6.40 Yes. When it is determined that the branch is taken (in WB), the pipeline will be flushed. At the same time, the 1 w instruction will stall the pipeline since the load value is not available for add. Both flush and stall will zero the control signals. The flush should take priority since the 1 w stall should not have occurred. They are on the wrong path. One solution is to add the flush pipeline signal to the Hazard Detection Unit. If the pipeline needs to be flushed, no stall will take place.

6.41 The store instruction can read the value from the register if it is produced at least 3 cycles earlier. Therefore, we only need to consider forwarding the results produced by the two instructions right before the store. When the store is in EX stage, the instruction 2 cycles ahead is in WB stage. The instruction can be either a 1 w or an ALU instruction.

```
assign EXMEMrt = EXMEMIR[ZO:16];

 assign bypassVfromWB - (IDEXop — SW) 5 CIOEXrt !- 0) &
    { ((MEMWBop — LW)   & (IDEXrt — HEMWBrt)) j
      ((MEMWBop —ALUop) & (IDEXrt — MEMWBrd)) );
```

This signal controls the store value that goes into EX/MEM register. The value produced by the instruction 1 cycle ahead of the store can be bypassed from the MEM/WB register. Though the value from an ALU instruction is available 1 cycle earlier, we need to wait for the load instruction anyway.

```
assign bypassVfromWB2 - (EXHEMop — SW) & (EXMEMrt !- 0) &
     (ibypassVfroinWB) &
   (  {{MEMWBop — LW)    & (EXMEMrt — MEMWBrt)) |
     {(MEMWBop — ALUop) & (EXMEMrt — MEMWBrd)) );
```

This signal controls the store value that goes into the data memory and MEM/WB register.

6.42

```
assign bypassAfromMEM - (IDEXrs 1- 0) &
  (  ((EXMEMop —- LW) & (IDEXrs — EXMEMrt)) |
    ((EXMEMop — ALUop) & (IDEXrs — EXMEMrd)) );
assign bypassAfromWB = (IDEXrs 1= 0) & (loypassAfromMEM) &
   (  ((MEMWBop — LW)    & (IDEXrs — MEMBrt)) |
     ((MEMWBop — ALUop) & (IDEXrs — MEMBrd)) ):
```

6.43 The branch cannot be resolved in ID stage if one branch operand is being calculated in EX stage (assume there is no dumb branch having two identical operands; if so, it is a jump), or to be loaded (in EX and MEM).

```
assign brandiStallinID = CIFIDop =- BEQ) &
   ( ((IOEXop — ALUop) S ({IFIDrs — IDEXrd) |
      (IFIDrt — IDEXrd)) ) | // alii in EX
     ((IDEXop — LW) & ((IFIDrs — IDEXrt) |
      (IFIDrt — IDEXrt)) ) | // lw in EX
     ((EXMEMop — LW) & ((IFIDrs — EXMEMrt) |
      (IFIDrt == EXMEMrt)) ) ); // lw in MEM
```

Therefore, we can forward the result from an ALU instruction in MEM stage, and an ALU or 1 w in WB stage.

```
assign bypassIDA = (EXMEMop — ALUop) & (IFIDrs — EXMEMrd);
assign bypassIDB = (EXMEMop — ALUop) & (IFIDrt — EXMEMrd);
```

Thus, the operands of the branch become the following:

```
assign IDAin =- bypassIDA ? EXMEMALUout : Regs[IFIDrs];
assign IDBTn - bypassIDB ? EXMEMALUout : Regs[IFIDrt];
```

And the branch outcome becomes:

```
assign takebranch = (IFIDop == BEQ) & (IDAin == IDBin);
```

5.44 For a delayed branch, the instruction following the branch will always be executed. We only need to update the PC after fetching this instruction.

```
If(-stall) begin IFIDIR <- IMemoryEPC]; PC <- PC+4; end;
if(takebranch) PC <- PC + (161IFIDIRC15]) +4; end;
```

6.45

```
module PredictPCfcurrentPC, nextPC, miss, update, destination);
input currentPC, update, desti nati on;
output nextPC, miss;
integer index, tag;
//512 entries, direct-map
reg[31:0] brTargetBuf[0:611], brTargetBufTag[O:511];
index = (currentPC>>2) & 511;
tag = currentPC»(2+9);
if(update) begin //update the destination and tag
  brTargetBuf[index]-destination;
  brTargetBufTag[index]=tag; end;
else if(tag==brTargetBufTag[index]) begin //a hit!
  nextPC-brTargetBuf[index]; miss-FALSE; end;

  else miss-TRUE:
endmodule;
```

6.46 No solution provided.

6.47

```
Loop:   lw   SZ. 0(510)
        lw   $5, 4(510)
        sub  $4, $2, $3
        sub  $6, $5, $3
        sw   $4, 0(S10)
        sw   S6. 4(510)
        addi $10, $10, 8
        bne  $10, $30, Loop
```

6.48 The code can be unrolled twice and rescheduled. The leftover part of the code can be handled at the end. We will need to test at the beginning to see if it has reached the leftover part (other solutions are possible.

```
Loop:       add!  $10,  $10.  12
            bgt   $10,  $30,  Leftover
            lw    $2.  -12($10)
            lw    $5,  -8<$10)
            lw    $7,  -4($10)
            sub   $4,  $2,  $3
            sub   $6,  $5,  $3
            sub   $8,  $7,  $3
            sw    $4,  -12($10)
            sw    $6,  -8($10)
            sw    $8,  -4($10)
            bne   $10,  $30,  Loop
            jump  Fini sh
Leftover:   lw    $2,  -12($10)
            sub   $4,  $2,  $3
            sw    $4,  -12($10)
            add!  $10,  $10,  -8
            beq   $10,  $30,  Firiish
            1 w   $5.  4($10)
            sub   $6.  $5,  $3
            sw    $6,  4($10)
Finish:  ...
```

6.49

```
       alu or branch                                hv/sw
Loop:       addi  $20 .  $10,  0          lw $2.  0($10)
                                          lw $5.  4{$10)
            sub   $4,    $2 .   $3        lw $7,  8($10)
            sub   $6,    $5 .   $3        lw $8,  12($10)
            sub   $11 ,  $7 .   $3        sw $4,  0($10)
            sub   $12 ,  $8 .   $3        sw $6,  4($10)
            addi  $10 ,  $10,  16         sw $11 ,  8($20)
            bne   $10 ,  $30,  Loop       sw $12 ,  12($20)
```

6.50 The pipe stages added for wire delays dcmot prodluce any useful work. Withi imperfect pipelining due to pipeline overhead, the overhead associated with these stages reduces throughput. These extra stages increase the control logic complexity since more pipe stages need to be covered. When considering penalties for branches mispredictions, etc., adding more pipe stages increase penalties and execution latency.

6.51 No solution provided.

Solutions for Chapter 7 Exercises

7.1 There are several reasons why you may not want to build large memories out of SRAM. SRAMs require more transistors to build than DRAMs and subsequently use more area for comparably sized chips. As size increases, access times increase and power consumption goes up. On the other hand, not using the standard DRAM interfaces could result in lower latency for the memory, especially for systems that do not need more than a few dozen chips.

7.2-7.4 The key features of solutions to these problems:

- Low temporal locality for data means accessing variables only once.
- High temporal locality for data means accessing variables over and over again.
- Low spatial locality for data means no marching through arrays; data is scattered.
- High spatial locality for data implies marching through arrays.

7.5 For the data-intensive application, the cache should be write-back. A write buffer is unlikely to keep up with this many stores, so write-through would be too slow.

For the safety-critical system, the processor should use the write-through cache. This way the data will be written in parallel to the cache and main memory, and we could reload bad data in the cache from memory in the case of an error.

7.9 2-miss, 3-miss, 11-miss, 16-miss, 21-miss, 13-miss, 64-miss, 48-miss, 19-miss, 11—hit, 3-miss, 22-miss, 4-miss, 27-miss, 6-miss, 11-set.

Cache set	Address
0000	4a
0001	
0010	2
0011	3
0100	4
0101	21
0110	6
0111	
1000	
1001	27
1010	
1011	11
1100	
1101	13
1110	
1111	

7.10 2-miss, 3—hit, 11-miss, 16-miss, 21-miss, 13-miss, 64-miss, 48-miss, 19-miss, 1 i—hit, 3-miss, 22-hit, 4-miss, 27-miss, 6-hit, 11-miss

Cache set	Address
00	[0,1, 2, 3]
01	[4, 5, 6, 7]
10	[8, 9. 10, 11]
11	[12. 13, 14, 15]

7.11 C stores multidimensional arrays in row-major form. Therefore accessing the array in row-major form will be fester since there will be a greater degree of temporal and spatial locality. Column-major form, on the other hand, will result in capacity misses if the block size is more than one word.

7.12 The Intrinsity caches are 16 KB caches with 256 blocks and 16 words per block. Data is 64 bytes = 512 bytes. The tag field is 18 bits (32 - (8 + 6}).

Total bits = 256 x {Data + Tag + Valid)

= 256 X (512 bits + 18 bits + 1 bit)

= 135,936 bits

7.13 Simply extend the comparison to include the valid bit as the high-order bit of the cache tag and extend the address tag by adding a high-order " 1 " bit. Now the values are equal only if the tags match and the valid bit is a 1.

7.14 The miss penalty is the time to transfer one block from main memory to the cache. Assume that it takes 1 clock cycle to send the address to the main memory.

a. Configuration (a) requires 16 main memory accesses to retrieve a cache block, and words of the block are transferred 1 at a time.

   Miss penalty = 1 + 16 x 10 + 16 x 1 = 177 clock cycles.

b. Configuration (b) requires 4 main memory accesses to retrieve a cache block and words of the block are transferred 4 at a time.

   Miss penalty = 1 + 4 x 10 + 4 x 1 = 45 clock cycles.

c. Configuration (c) requires 4 main memory accesses to retrieve a cache block, and words of the block are transferred 1 at a time.

   Miss penalty = 1 + 4 x 10 + 16 x 1 = 57 clock cycles

7.16 The shortest reference string will have 4 misses for Cl and 3 misses for C2. This leads to 32 miss cycles versus 33 miss cycles. The following reference string will do: 0x00000000, 0x00000020,0x00000040, 0x00000041.

7.17 AMAT = Hit time + Miss rate x Miss penalty

AMAT = 2 ns + 0.05 x (20 x 2ns) = 4 ns

7.18 AMAT = (1.2 x 2 ns) + (20 x 2 ns x 0.03) = 2.4 ns + 1.2 ns = 3.6 ns

Yes, this is a good choice.

7.19 Execution time = Clock cycle x IC x (CPI^ + Cache miss cycles per instruction)

Execution $\text{time}_{\text{original}}$ = 2 x IC x (2 + 1.5 x 20 x 0.05) = 7 IC

Execution time_{new} = 2.4 x IC x (2 + 1.5 x 20 x 0.03) = 6.96 IC

Hence, doubling the cache size to improve miss rate at the expense of stretching the clock cycle results in essentially no net gain.

7.20

Reference	Band	Bank Conflict
3	3	No
9	1	No
17	1	Yes (with 9)
2	2	No
51	3	No
37	1	Yes (with 17)
13	1	Yes (with 37
4	0	No
a	0	Yes (with 4)
41	1	No
67	3	No
10	2	NO

A bank conflict causes the memory system to stall until the busy bank has completed the prior operation.

7.21 No solution provided.

7.22 No solution provided.

7.28 Two principles apply to this cache behavior problem. First, a two-way set-associative cache of the same size as a direct-mapped cache has half the number of sets. Second, LRU replacement can behave pessimally (as poorly as possible) for access patterns that cycle through a sequence of addresses that reference more blocks than will fit in a set managed by LRU replacement.

Consider three addresses—call them A, B, C—that all map to the same set in the two-way set-associative cache, but to two different sets in the direct-mapped cache. Without loss of generality, let A map to one set in the direct-mapped cache and B and C map to another set. Let the access pattern be A B C A B C A . . . and so on. The direct-mapped cache will then have miss, miss, miss, hit, miss, miss, hit, . . . , and so on. With LRU replacement, the block at address C will replace the block at the address A in the two-way set-associative cache just in time for A to be referenced again. Thus, the two-way set-associative cache will miss on every reference as this access pattern repeats.

7.29

Address size:	it bits
Cache size:	*S* bytes/cache
Block size:	$B = 2^b$ bytes/block
Associativity:	A blocks/set

Number of sets in the cache:

$$\text{Sets/cache} = \frac{\text{(Bytes/cache)}}{\text{(Blocks/set) X (Bytes/block)}}$$

$$= \frac{S}{AB}$$

Number of address bits needed to index a particular set of the cache:

$$\text{Cache set index bits} = \log_2 \text{(Sets/cache)}$$

$$= \log_2\left(\frac{S}{AB}\right)$$

$$= \log_2\left(\frac{S}{A}\right) - b$$

Number of bits needed to implement the cache:

Tag address bits/block = (Total address bits) - (Cache set index bits)

- {Block offset bits)

$$= k - \left(\log_2\left(\frac{S}{A}\right) - b\right) - b$$

—(I)

Number of bits needed to implement the cache = sets/cache x associativity x (data + tag + valid):

$$=_{AB}\frown\cdots\frown\ 8 \times B + k - \log_2\left(\frac{S}{A}\right) + 1\Big)$$

$$= \frac{S}{B} \times \left(8B + k - \log_2\left(\frac{S}{A}\right) + 1\right)$$

7.32 Here are the cycles spent for each cache:

Cache	Miss penalty	I cache miss	D cache miss	tSSm miss
Cl	6 + 1 = 7	4% x 7 = 0.28	6% x 7 = 0.42	$0.28 \times \frac{0.42}{2} = 0.49$
C2	6 + 4 = 10	*2% x* 10 = 0.20	4% x 10 = 0.4	$0.20 \times \frac{U.4}{2} = 0.4$
C3	8 + 4-10	2% x 10 = 0.20	3% x 10 = 0.3	$0.20 \times \frac{0.3}{2} = 0.35$

Therefore Cl spends the most cycles on cache misses.

7.33 Execution time = CPI x Clock cycle x IC

We need to calculate the base CPI that applies to all three processors. Since we are given CPI = *2* for Cl,

CPI_base = CPI - CPI^^ = 2 - 0.49 = 1.51

ExCl = 2 x 420 ps x IC = 8.4 x $10''^{10}$ x IC

ExC2 = (1.51 + 0.4) X 420 ps X IC = 8.02 x $10''^{10}$ X IC

ExC3 = (1.51 + 0.35) X 310 ps x IC = 5.77 X $10''^{10}$ X IC

Therefore C3 is fastest and Cl is slowest.

7.34 No solution provided.

7.35 If direct mapped arid stride = 256, then we can assume without loss of generality that array[0]... array[31] is in block 0. Then, block I has array [32]... [63] and block 2 has array[64] .. . [127] and so on until block 7 has [224] . . . [255]. (There are 8 blocks X 32 bytes = 256 bytes in the cache.) Then wrapping around, we find also that block 0 has array[256]... [287], and so on.

Thus if we look at array[0] and array[256], we are looking at the same cache block. One access will cause the other to miss, so there will be a 100% miss rate. If the stride were 255, then array [0] would map to block 0 while array [255] and array [510] would both map to block 7. Accesses to array [0] would be hots, and accesses to array [255] and array [510] would conflict. Thus the miss rate would be 67%.

If the cache is two-way set associative, even if two accesses are in the same cache set, they can coexist, so the miss rate will be 0.

7.38 No solution provided.

7.39 The total size is equal to the number of entries times the size of each entry. Each page is 16 KB, and thus, 14 bits of the virtual and physical address will be used as a page offset. The remaining 40 - 14 = 26 bits of the virtual address constitute the virtual page number, and there are thus 2^{26} entries in the page table, one for each virtual page number. Each entry requires 36 - 14 = 22 bits to store the physical page number and an additional 4 bits for the valid, protection, dirty, and use bits. We round the 26 bits up to a full word per entry, so this gives us a total size of 2^{26} x 32 bits or 256 MB.

7.40 No solution provided.

7.41 The TLB will have a high miss rate because it can only access 64 KB (16 x 4 KB) directly. Performance could be improved by increasing the page size if the architecture allows it.

7.42 Virtual memory can be used to mark the heap and stack as read and write only. In the above example, the hardware will not execute the malicious instructions because the stack memory locations are marked as read and write only and not execute.

7.45 Less memory—fewer compulsory misses. (Note that while you might assume that capacity misses would also decrease, both capacity and conflict misses could increase or decrease based on the locality changes in die rewritten program. There is not enough information to definitively state the effect on capacity and conflict misses.)

Increased clock rate—in general there is no effect on the miss numbers; instead, the miss penalty increases. (Note for advanced students: since memory access timing would be accelerated, there could be secondary effects on the miss numbers if the hardware implements prefetch or early restart.)

Increased associativity—fewer conflict misses.

7.46 No solution provided.

7.47 No solution provided.

7.48 No solution provided.

7.49 No solution provided.

7.50 No solution provided.

7.51 No solution provided.

7.52 This optimization takes advantage of spatial locality. This way we update all of the entries in a block before moving to another block.

Solutions for Chapter 8 Exercises

8.1 Each transaction requires 10,000 x 50 = 50,000 instructions.

CPU limit: 500M/50K = 10,000 transactions/second.

The I/O limit for A is 1500/5 = 300 transactions/second.

The I/O limit for B is 1000/5 = 200 transactions/second.

These I/O limits limit the machine.

8.2 System A

transactions	9	compute	1	
I/Os	45	latency	5	
times	900 ms	100 us	100 ins	exceeds 1 s

Thus system A can only support 9 transactions per second.

System B—first 500 I/Os (first 100 transactions)

transactions	9	compute	1	1	
l/0s	45	latency	5	5	
times	810 ms	100 us	90 ms	90 ms	990.1ms

Thus system B can support 11 transactions per second at first.

8.3 Time/file = 10 seconds + 40 MB * 1/(5/8) seconds/MB - 74 seconds

Power/file = 10 seconds * 35 watts + (74 - 10) seconds * 40 watts = 2910 I

Number of complete files transferred = 100,000 J/2910 J = 34 files

8.4 Time/file = 10 seconds + 0.02 seconds + 40 MB * 1/(5/8) seconds/MB = 74.02 seconds

Hard disk spin time/file = 0.02 seconds + 40 MB * 1/50 seconds/MB = 0.82 seconds

Power/file - 10 seconds * 33 watts + 0.02 seconds * 38 watts + 0.8 seconds * 43 watts + 63.2 seconds * 38 watts = 330 J + 0.76 J + 34.4 J + 2401.6 J = 2766.76 J

Number of complete files transferred = 100000 J / 2766.761 = 36 files

Energy for all 100 files = 2766.76 * 100 = 276676 J

8.5 After reading sector 7, a seek is necessary to get to the track with sector 8 on it. This will take some time (on the order of a millisecond, typically), during which the disk will continue to revolve under the head assembly. Thus, in the version where sector 8 is in the same angular position as sector 0, sector S will have already revolved past the head by the time the seek is completed and some large fraction of an additional revolution time will be needed to wait for it to come back again. By skewing the sectors so that sector 8 starts later on the second track, the seek will have time to complete, and then the sector will soon thereafter appear under the head without the additional revolution.

8.6 No solution provided.

8.7

a. Number of heads =15
b. Number of platters = 8
c. Rotational latency = 8.33 ms
d. Head switch time = 1.4 ms
e. Cylinder switch time = 2.1 ms

8.8

a. System A requires 10 + 10 - 20 terabytes.

   System B requires 10 + 10 * 1/4 = 12,5 terabytes.

   Additional storage: 20 - 12.5 = 7.5 terabytes.
b. System A: 2 blocks written = 60 ms.

   System B: 2 blocks read and written = 120 ms.
c. Yes. System A can potentially accommodate more failures since it has more redundant disks. System A has 20 data disks and 20 check disks. System B has 20 data disks and 5 check disks. However, two failures in the *same* group will cause a loss of data in both systems.

8.9 The power failure could result in a parity mismatch between the data and check blocks. This could be prevented if the writes to the two blocks are performed simultaneously,

8.10 20 meters time: 20 m * 1/(1.5 * 10^{8}) s/m = 133.3 ns

2,000,000 meters time: 2000000 m * 1/(1.5 * 10^{8}) s/m = 13.3 ms

8.11 20 m: 133.3 * $10''^{9}$ s * 6 MB/sec = 0.8 bytes

2000000 m: 13.3 * $10''^{3}$ s * 6 MB/sec = 80 KB

8.12 4 KHz * 2 bytes/sample * 100 conversations = 800,000 bytes/sec

Transmission time is 1 KB/5 MB/sec + 150 us = 0.00035 seconds/KB

Total time/KB = 800 * 0.00035 = 0.28 seconds for 1 second of monitoring

There should be sufficient bandwidth.

8.13

a. 0

b. 1

c. 1

d. 2

e. Each bit in a 3-bit sequence would have to be reversed. The percentage of errors is 0.01 "0.01 *0.01 =0.000001 (or 0.0001%)

8.14

a. 1

b. 0

8.15

a. Not necessarily, there could be a single-bit error or a triple-bit error.

b. No. Parity only specifies whether an error is present, not which bit the error is in.

c. No. There could be a double-bit error or the word could be correct.

8.16 (Seek time + Rotational delay + Overhead) * 2 + Processing time

(0.008 sec + 0.5 / (10000/60) sec + 0.002) * 2 + (20 million cyctes)(5 GHz) sec = (.008 + .003 + .002)*2 + .004 = 30 ms

Block processed/second = 1/30 ms = 33.3

Transfer time is 80 usec and thus is negligible.

8.17 Possible answers may include the following:

- Application programmers need not understand how things work in lower levels.
- Abstraction prevents users from making low-level errors.
- Flexibility: modifications can be made to layers of the protocol without disrupting other layers.

8.1S For 4-word block transfers, the bus bandwidth was 71.11 MB/sec. For 16-word block transfers, the bus bandwidth was 224.56 MB/sec. The disk drive has a transfer rate of 50 MB/sec. Thus for 4-word blocks we could sustain 71/50 = 1 simultaneous disk transfers, and for 16-word blocks we could sustain 224/50 = 4 simultaneous disk transfers. The number of simultaneous disk transfers is inherently an integer and we want the sustainable value. Thus, we take the floor of the quotient of bus bandwidth divided by disk transfer rate.

8.19 For the 4-word block transfers, each block now takes

1. 1 cycle to send an address to memory
2. 150 ns/5 ns = 30 cycles to read memory
3. 2 cycles to send the data
4. 2 idle cycles between transfers

This is a total of 35 cycles, so the total transfer takes 35 x 64 = 2240 cycles. Modifying the calculations in the example, we have a latency of 11,200 ns, 5.71M transactions/second, and a bus bandwidth of 91.43 MB/sec.

For the 16-word block transfers, each block now takes

1. 1 cycle to send an address to memory
2. 150 ns or 30 cycles to read memory
3. 2 cycles to send the data
4. 4 idle cycles between transfers, during which the read of the next block is completed

Each of the next two remaining 4-word blocks requires repeating the last two steps. The last 4-word block needs only 2 idle cycles before the next bus transfer. This is a total of 1 + 20 -f 3 * (2 + 4) + (2 + 2) = 53 cycles, so the transfer takes 53 * 16 = 848 cycles. We now have a latency of 4240 ns, 3.77M transactions/second, and a bus bandwidth of 241.5 MB/sec.

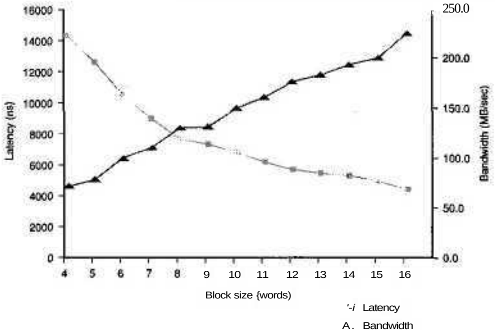
Note that the bandwidth for the larger block size is only 2.64 times higher given the new read times. This is because the 30 ns for subsequent reads results in fewer opportunities for overlap, and the larger block size performs (relatively) worse in this situation.

8.20 The key advantage would be that a single transaction takes only 45 cycles, as compared with 57 cycles for the larger block size. If because of poor locality we were not able to make use of the extra data brought in, it might make sense to go with a smaller block size. Said again, the example assumes we want to access 256 words of data, and dearly larger block sizes will be better. (If it could support it, we'd like to do a single 256-word transaction!)

8.21 Assume that only the 4-word reads described in the example are provided by the memory system, even if fewer than 4 words remain when transferring a block. Then,

	Block size (words)												
	4	5	6	7	8	9	10	11	12	13	14	15	16
Number of 4-word transfers to send the block	1	2	2	2	2	3	3	3	3	4	4	4	4
Time to send address to memory (bus cycles)	1	1	1	1	1	1	1	1	1	1	1	1	1
Time to redd first 4 words in memory (bus cycles)	40	40	40	40	40	40	40	40	40	40	40	40	40
Block trensfer time$_p$ at 2 transfer bus cycles and 2 idle bus cycles per 4-word transfer (bus cycles)	4	8	8	8	8	12	12	12	12	16	16	16	16
Total time to transfer one block (bus cycles)	45	49	49	49	49	S3	53	53	53	57	57	57	57
Number of bus transactions to read 256 words using the given block size	64	52	43	37	32	29	26	24	22	20	19	18	16
Time for 256-word transfer (bus cycles)	2880	2548	2107	1813	1568	1537	1378	1272	1166	1140	1083	1026	912
Latency (IK)	14400	12740	10535	9065	7840	7685	6890	6360	5830	5700	5415	5130	4560
Number of bus transactions (millions per second)	4.444	4.082	4.082	4.082	4.082	3.774	3.774	3.774	3.774	3.509	3.509	3.509	3.509
Bandwidth (MB/«oc)	71.1	80.4	97.2	113.0	130.6	133.2	148.6	161.0	175.6	179.6	189.1	199.6	224.6

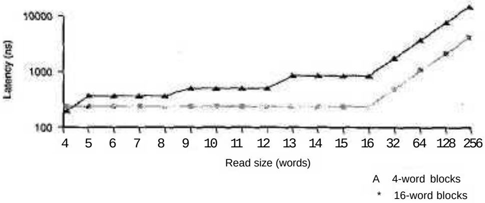
The following graph plots latency and bandwidth versus block size:



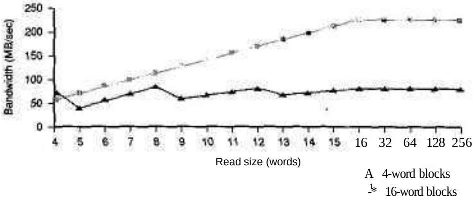
8.22 From the example, a 4-word transfer takes 45 bus cycles and a 16-word block transfer takes 57 bus cycles. Then,

	Read size (words)																
	4	5	6	7	8	9	10	11	12	13	14	15	16	32	64	128	256
Number of 4-worO transfers to send the data	1	2	2	2	2	3	3	3	3	4	4	4	4	**a**	16	32	64
Number of 16-word transfers to send the data	1	1	1	1	1	1	1	1	1	1	1	1	1	2	4	8	16
Total read time using 4-word blocks (bus cycles)	45	90	90	90	90	135	135	135	135	180	180	180	180	360	720	1440	2880
Total read time using 16-word blocks (bus cycles)	57	57	57	57	57	57	57	57	57	57	57	57	57	114	228	456	912
Latency using 4-word block* <ns>	225	450	450	450	450	675	675	675	675	900	900	900	900	1800	3600	7200	14400
Latency n-Inf le-word block* (ns)	285	285	285	285	285	285	285	285	285	285	285	285	285	570	1140	2280	4560
Bandwidth using 4-word blocks (MB/MC)	71.1	44.4	53.3	62.2	71.1	53.3	59.3	65.2	71.1	57.8	62.2	66.7	71.1	71.1	71.1	71.1	71.1
Bandwidth using 18-word blocks (MB/soc)	56.1	70.2	84.2	98.2	112.3	126.3	140.4	154.4	168.4	182.5	196.5	210.5	224.6	224.6	224.6	224.6	224.6

The following graph plots read latency with 4-word and 16-word blocks:



The following graph plots bandwidth with 4^word and 16-word blocks:



8.23

For 4-word blocks:

Send address and first word simultaneously = I clock
Time until first write occur = 40 clocks
Time to send remaining 3 words over 32-bit bus = 3 clocks
Required bus idle time = 2 clocks
Total time = 46 clocks

Latency = 64 4-word blocks at 46 cycles per block = 2944 clocks = 14720 ns
Bandwidth = (256 x 4 bytes)/14720 ns = 69.57 MB/sec

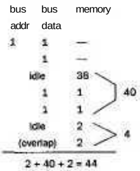
For 8-word blocks:

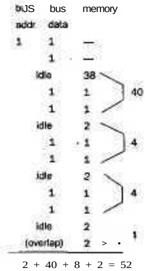
Send address and first word simultaneously =	1 clock
Time until first write occurs =	40 clocks
Time to send remaining 7 words over 32-bit bus =	7 clocks
Required bus idle time (two idle periods) =	4 docks
Total time =	52 clocks

Latency = 32 8-word blocks at 52 cycles per block = 1664 clocks = 8320 ns

Bandwidth = (256 x 4 bytes)/8320 ns = 123.08 MB/sec

In neither case does the 32-bit address/32-bit data bus outperform the 64-bit combined bus design. For smaller blocks, there could be an advantage if the overhead of a fixed 4-word block bus cycle could be avoided.


4-word transfer*


8-word transform

8.24 For a 16-word read from memory, there will be four sends from the 4-word-wide memory over the 4-word-wide bus. Transactions involving more than one send over the bus to satisfy one request are typically called *burst transactions.*

For burst transactions, some way must be provided to count the number of sends so that the end of the burst will be known to all on the bus. We don't want another device trying to access memory in a way that interferes with an ongoing burst transfer. The common way to do this is to have an additional bus control signal, called BurstReq or Burst Request, that is asserted for die duration of the burst.

This signal is unlike the ReadReq signal of Figure 8.10, which is asserted only long enough to start a single transfer. One of the devices can incorporate the counter necessary to track when BurstReq should be deasserted, but both devices party to the burst transfer must be designed to handle the specific burst (4 words, 8 words, or other amount) desired. For our bus, if BurstReq is not asserted when ReadReq signals the start of a transaction, then the hardware will know that a single send from memory is to be done.

So the solution for the 16-word transfer is as follows: The steps in the protocol begin immediately after the device signals a burst transfer request to the memory by raising ReadReq and Burst_Request and putting the address on the Date lines.

1. When memory sees the ReadReq and BurstReq lines, it reads the address of the start of the 16-word block and raises Ack to indicate it has been seen.
2. I/O device sees the Ack line high and releases the ReadReq and Data lines, but it keeps BurstReq raised.
3. Memory sees that ReadReq is low and drops the Ack line to acknowledge the ReadReq signal.
4. This step starts when BurstReq is high, the Ack line is low, and the memory has the next 4 data words ready. Memory places the next 4 data words in answer to the read request on the Data lines and raises DataRdy.
5. The I/O device sees DataRdy, reads the data from the bus, and signals that it has the data by raising Ack.
6. The memory sees the Ack signal, drops DataRdy, and releases the Data lines.
7. After the I/O device sees DataRdy go low, it drops the Ack line but continues to assert BurstReq if more data remains to be sent to signal that it is ready for the next 4 words. Step 4 will be next if BurstReq is high.
8. If the last 4 words of the 16-word block have been sent, the I/O device drops BurstReq, which indicates that the burst transmission is complete.

With handshakes taking 20 ns and memory access taking 60 ns, a burst transfer will be of the following durations:

Step 1 20 ns (memory receives the address at the end of this step; data goes on the bus at the beginning of step 5)

Steps 2,3,4 Maximum (3 x 20 ns, 60 ns) = 60 ns

Steps 5,6,7,4 Maximum (4 x 20 ns, 60 ns) = 80 ns (looping to read and then send the next 4 words; memory read latency completely hidden by handshaking time)

Steps 5,6, 7,4 Maximum {4 x 20 ns, 60 ns) = 80 ns (looping to read and then send the next 4 words; memory read latency completely hidden by handshaking time)

Steps 5, 6,7, 4 Maximum (4 x 20 ns, 60 ns) = 80 ns {looping to read and then send the next four words; memory read latency completely hidden by handshaking time)

End of burst transfer

Thus, the total time to perform the transfer is 320 ns, and the maximum bandwidth is

(16 words x 4 bytes)/320 ns = 200 MB/sec

It is a bit difficult to compare this result to that in the example on page 665 because the example uses memory with a 200 ns access instead of 60 ns. If the slower memory were used with the asynchronous bus, then the total time for the burst transfer would increase to 820 ns, and the bandwidth would be

(16 words X 4 bytes)/820 ns = 78 MB/sec

The synchronous bus in the example on page 665 needs 57 bus cycles at 5 ns per cycle to move a 16-word block. This is 285 ns, for a bandwidth of

(16 words x 4 bytes)/285 ns = 225 MB/sec

8.26 No solution provided

8.27 First, the synchronous bus has 50-ns bus cycles. The steps and times required for the synchronous bus are as follows:

Send the address to memory: 50 ns

Read the memory: 200 ns

Send the data to the device: 50 ns

Thus, the total time is 300 ns. This yields a maximum bus bandwidth of 4 bytes every 300 ns, or

$$\frac{4 \text{ bytes}}{300 \text{ ns}} = \frac{4\text{MB}}{0.3 \text{ seconds}} \sim 13.3\frac{\text{MB}}{\text{second}}$$

At first glance, it might appear that the asynchronous bus will be *much* slower, since it will take seven steps, each at least 40 ns, and the step corresponding to the memory access will take 200 ns. If we look carefully at Figure 8.10, we realize that several of the steps can be overlapped with the memory access time. In particular, the memory receives the address at the end of step 1 and does not need to put the data on the bus until the beginning of step 5; steps 2,3, and 4 can overlap with the memory access time. This leads to the following timing:

Step 1: 40 ns

Steps 2,3,4: maximum (3 x 40 ns, 200 ns) = 200 ns

Steps5,*6,7:* 3X40ns = 120ns

Thus, the total time to perform the transfer is 360 ns, and the maximum bandwidth is

$$\frac{\text{4bytes}}{\text{360 ns}} = \frac{\text{4MB}}{\text{0.36 seconds}} = 11.1\ \frac{\text{MB}}{\text{second}}$$

Accordingly, the synchronous bus is only about 20% faster. Of course, to sustain these rates, the device and memory system on the asynchronous bus will need to be fairly fast to accomplish each handshaking step in 40 ns.

8.28 For the 4-word block transfers, each block takes

1. 1 clock cycle that is required to send the address to memory
2. $\frac{200ns}{\text{5 ns/cyde}}$ = 40 dock cycles to read memory
3. 2 clock cycles to send the data from the memory
4. 2 idle clock cydes between this transfer and the next

This is a total of 45 cydes, and 256/4 = 64 transactions are needed, so the entire transfer takes 45 X 64 = 2880 dock cycles. Thus the latency is 2880 cydes X 5 ns/cyde = 14,400 ns.

Sustained bandwidth is $\frac{256}{14}$ ^ t y $\frac{e\ S}{ns}$ =71.11 MB/sec.

The number of bus transactions per second is

$$\frac{\text{64 transactions}}{\text{14,400 ns}} = 4.44 \text{ transactions/second}$$

For the 16-word block transfers, the first block requires

1. 1 dock cycle to send an address to memory
2. 200 ns or 40 cydes to read the first four words in memory
3. 2 cycles to send the data of the block, during which time the read of the four words in the next block is started
4. 2 idle cycles between transfers and during which the read of the next block is completed

Each of the three remaining 16-word blocks requires repeating only the last two steps.

Thus, the total number of cycles for each 16-word block is 1 + 40 + 4 X (2 + 2) = 57 cycles, and 256/16 = 16 transactions are needed, so the entire transfer takes, 57 x 16 = 912 cycles. Thus the latency is 912 cycles x 5 ns/cyde = 4560 ns, which is roughly one-third of the latency for the case with 4-word blocks.

Sustained bandwidth is $\frac{256 \times 4 \text{ bytes}}{4560 \text{ ns}} = 224.56$ MB/sec

The number of bus transactions per second with 16-word blocks is

$$\frac{16 \text{ transactions}}{4560 \text{ ns}} = 3.51\text{M transactions/second}$$

which is lower than the case with 4-word blocks because each transaction takes longer (57 versus 45 cydes).

8.29 First the mouse:

Clock cydes per second for polling = 30 x 400 = 12,000 cydes per second

$$\text{Fraction of the processor dock cycles consumed} = \frac{12 \times 10^3}{500 \times 10^6} = 0.002\%$$

Polling can dearly be used for the mouse without much performance impact on the processor.

For the floppy disk, the rate at which we must poll-is

$$\frac{50 \frac{\text{KB}}{\text{second}}}{2 \frac{\text{bytes}}{\text{polling access}}} = 25\text{K} \frac{\text{polling accesses}}{\text{second}}$$

Thus, we can compute the number of cycles:

$$\text{Cycles per second for polling} = 25\text{K} \times 400 = 10 \times 10^6$$

$$\text{Fraction of the processor consumed} = \frac{10 \times 10^6}{500 \times 10^6} = 2\%$$

This amount of overhead is significant, but might be tolerable in a low-end system with only a few I/O devices like this floppy disk.

In the case of the hard disk, we must poll at a rate equal to the data rate in four-word chunks, which is 250K times per second (4 MB per second/16 bytes per transfer). Thus,

$$\text{Cycles per second for polling} = 250\text{K} \times 400$$

$$\text{Fraction of the processor consumed} = \frac{100 \times 10\wedge}{500 \times \text{LO}^6} = 20\%$$

Thus one-fifth of the processor would be used in just polling the disk. Clearly, polling is likely unacceptable for a hard disk on this machine.

8.30 The processor-memory bus takes 8 clock cycles to accept 4 words, or 2 bytes/clock cycle. This is a bandwidth of 1600 MB/sec. Thus, we need 1600/40 = 40 disks, and because all 40 are transmitting, we need 1600/100 = 16 I/O buses.

8.31 Assume the transfer sizes are 4000 bytes and 16000 bytes (four sectors and sixteen sectors, respectively). Each disk access requires 0.1 ms of overhead + 6 ms of seek.

For the 4 KB access (4 sectors):

- Single disk requires 3 ms + 0.09 ms (access time) +6.1 ms = 9.19 ms
- Disk array requires 3 ms + 0.02 ms (access time) + 6.1 ms = 9.12 ms

For the 16 KB access (16 sectors):

- Single disk requires 3 ms + 0.38 ms (access time) + 6.1 ms = 9.48 ms
- Disk array requires 3 ms + 0.09 ms (access time) + 6.1 ms = 9.19 ms

Here are the total times and throughput in I/Os per second:

- Single disk requires (9.19 + 9.48)/2 = 9.34 ms and can do 107.1 I/Os per second.
- Disk array requires (9.12 + 9.19)/2 = 9.16 ms and can do 109.1 I/Os per second.

8.32 The average read is (4 + 16)/2 = 10 KB. Thus, the bandwidths are

Single disk: 107.1 * 10KB - 1071 KB/second.

Disk array: 109.1 * 10 KB = 1091 KB/second.

8.33 You would need I/O equivalents of Load and Store that would specify a destination or source register and an I/O device address (or a register holding the address). You would either need to have a separate I/O address bus or a signal to indicate whether the address bus currently holds a memory address or an I/O address.

8.34

a. If we assume that the processor processes data before polling for the next byte, the cycles spent polling are 0.02 ms * 1 GHz - 1000 cycles = 19,000 cycles. A polling iteration takes 60 cycles, so 19,000 cycles = 316.7 polls. Since it takes an entire polling iteration to detect a new byte, the cycles spent polling are 317 * 60 = 19,020 cycles. Each byte thus takes 19,020 + 1000 = 20,020 cycles. The total operation takes 20,020 * 1000 = 20,020,000 cycles.

   (Actually, every third byte is obtained after only 316 polls rather than 317; so, the answer when taking this into account is 20,000,020 cycles.)

b. Every time a byte comes the processor takes 200 + 1000= 1200 cycles to process the data. 0.02 ms * 1 GHz - 1200 cycles = 18,800 cycles spent on the other task for each byte read. The total time spent on the other task is 18,800 "1000= 18,800,000 cycles.

8.38 Some simplifying assumptions are the following:

- A fixed overhead for initiating and ending the DMA in units of clock cycles. This ignores memory hierarchy misses adding to the time.
- Disk transfers take the same time as the time for the average size transfer, but the average transfer size may not well represent the distribution of actual transfer sizes.
- Real disks will not be transferring 100% of the time—far from it.

Network: (2 us + 25 us * 0.6)/(2 us + 25 us) = 63% of original time (37% reduction)

Reducing the trap latency will have a small effect on the overall time reduction

8.39 The interrupt rate when the disk is busy is the same as the polling rate. Hence,

Cycles per second for disk = 250K x 500 = 125 x 10^6 cycles per second

Fraction of the processor consumed during a transfer =" $\frac{125 \times 10^6}{500x\sim Tb\sim«\sim}$ 25%

Assuming that the disk is only transferring data 5% of the time,

Fraction of the processor consumed on average = 25%x5%= 1.25%

As we can see, the absence of overhead when an I/O device is not actually transferring is the major advantage of an interrupt-driven interface versus polling.

8.40 Each DMA tranfer takes

$$\frac{8\text{ KB}}{4\dfrac{\text{MB}}{\text{second}}} = 2 \times 10^{-3}\text{ seconds}$$

So if the disk is constantly transferring, it requires

$$\frac{1000 + 500\dfrac{\text{cycles}}{\text{transfer}}}{2 \times 10^{-3}\dfrac{\text{seconds}}{\text{transfer}}} = 750 \times 10^{3}\ \frac{\text{clock Cycles}}{\text{second}}$$

Since the processor runs at 500 MHz,

$$\text{Fraction of processor consumed} = \frac{750 \times 10^{3}}{500 \times 10^{6}} = 1.5 \times 10^{-3} = 0.15\%$$

8.44 Maximum I/O rate of the bus: 1,000,000,000/8000 = 125,000 I/O/second

CPU bottleneck restricts throughput to 10,000 I/O / second
Time/I/O is 6.11 ms at disk, each disk can complete 1000/6.11 = 163.67 I/O/second

To saturate the CPU requires 10,000 I/O second.

$$\frac{10{,}000}{163.67} = 61\text{ disks.}$$

$$\frac{61\text{ disks}}{7\text{ disks/scsl controller}} = 9\text{ scsl controllers.}$$

8.45 First, check that neither the memory nor the I/O bus is a bottleneck. Sustainable bandwidth into memory is 4 bytes per 2 clocks = 800 MB/sec The I/O bus can sustain 100 MB/sec. Both of these are faster than the disk bandwidth of 40 MB/sec, so when the disk transfer is in progress there will be negligible additional time needed to pass the data through the I/O bus and write it into memory. Thus, we ignore this time and focus on the time needed by the DMA controller and the disk. This will take 0.1 ms to initiate, 8 ms to seek, 16 KB/40 MB to transfer: total = 8.5 ms.

8.46 Disk access total time: 10,000/500,000,000 s + 20 ms = 20.002 ms

% delay trapping to OS: 0.01%

Network access total time: 10,000/5,000,000,000 s + 25 us = 27 us

% delay trapping to OS: 7.4%

8.47 Disk: (2 us + 20 ms * 0.4)/(2 [is + 20 ms) = 40% of original time (60% reduction)

Reducing the trap latency will have virtually no effect on the overall time reduction

Network: (2 us + 25 us * 0.6)/(2 us + 25 us) = 63% of original time (37% reduction)

Reducing the trap latency will have a small effect on the overall time reduction

Solutions for Chapter 9 Exercises

9.1 No solution provided.

9.2 The short answer is that x is always 2, and y can either be 2,4, or 6. In a load-store architecture the code might look like the following:

Processor 1	Processor 2
load X Into a register	load X into a register
Increment register	Incrernent register
store register back to X	store register bach to Y
load Y into a register	
add two registers to register	
store register back to Y	

When considering the possible interleavings, only the loads/stores are really of interest. There are four activities of interest in process 1 and two in process 2. There are 15 possible interleavings, which result in the following:

111122: x = 2,y = 4
111212: x = 2,y =4
111221:x = 2,y = 2
112112: X =2,y =4
112121:x = 2,y = 2
112211:x = 2,y=6
121112:x=2,y = 2
121121:x = 2,y = 2
121211:x = 2,y=4
122111:x = 2,y=4
211112:x = 2,y = 2
211121:x = 2,y=2
211211: x=2,y =4
2121U:x = 2,y = 4
2211H:x = 2,y=4

9.3 No solution provided.

9.4 No solution provided.

9.5 Write-back cache with write-update cache coherency and one-word blocks. Both words are in both caches and are initially clean. Assume 4-byte words and byte addressing.

Total bus transactions = 2

	Action	Comment
1	PI writes to 100	One bus transfer to move the word at 100 from PI to P2 cache.
2	P2 writes to 104	One bus transfer to move the word at 104 from P2 to PI cache.
3	Pi reads 100	No bus transfer; word read from PI cache.
4	P2 reads 104	No bus transfer; word read from P2 cache.
5	PI reads 104	No bus transfer; word read from PI cache.
6	P2 reads 100	No bus transfer; won) read from P2 cache.

9.6 Write-back cache with write-update cache coherency and four-word blocks. The block is in both caches and is initially clean. Assume 4-byte words and byte addressing. Assume that the bus moves one word at a time. Addresses 100 and 104 are contained in the one block starting at address 96.

Step	Action	Comment
1	PI writes to 100	One bus transfer to move the word at 100 from PI to P2 cache.
2	P2 writes to 104	One bus transfer to move the word at 104 from P21a PI cache.
3	Plreaids 100	No bus transfer; word read from PI cache.
4	P2reaids 104	No bus transfer; word read from P2 cache.
5	Plreeids 104	No bus transfer; word read from PI cache.
6	P2resdsIOO	No bus transfer; word read from P2 cache.

Total bus transactions = 2.

False-sharing appears as a performance problem under a write-invalidate cache coherency protocol. Writes from different processors to different words that happen to be allocated in the same cache block cause that block to ping-pong between the processor caches. That is, a dirty block can reside in only one cache at a time.

If we modify the cache in this exercise to use write-invalidate, the number of bus transactions increases to 9.

Step	Action	Comment
1	PI writes 100	PI issues a write-invalidate using one bus transaction to send the address 100; P2 removes the block from its cache; the block Is now dirty In the PI cache alone.
2	P2 writes 104	P2 issues a read-with-intent-to-modify and the block Is moved from PI to P2 using four bus transfers; PI removes the block from Its cache; the block is now dirty in the P2 cache alone
3	PI reads 100	PI Issues a read miss and the P2 cache supplies the block to PI and writes back to the memory at the same time using four bus transfers; the block Is now clean in both cases.
4	P2 reads 104	No bus transfer; word read from P2 cache.
5	PI reads 104	No bus transfer; word read from PI cache.
6	P2 reads 100	No bus transfer; word read from P2 cache.

9.7 No solution provided.

9.8 No solution provided.

9.9 No solution provided.

9.10 No solution provided.

9.11 No solution provided.

9.12 No solution provided.

9.13 No solution provided.

9.14 No solution provided.

Solutions for Appendix B Exercises

B.1

A	B	$\overline{A}$	$\overline{B}$	$\overline{A+B}$	$\overline{A}\cdot\overline{B}$	$\overline{A\cdot B}$	$\overline{A}+\overline{B}$
0	0	1	1	1	1	1	1
0	1	1	0	**0**	0	1	1
1	0	0	1	0	0	1	1
1	1	0	0	0	**0**	**0**	0

B.2 Here is the first equation:

$$E = ((A \bullet B) + (A \bullet C) + (B \bullet C)) \bullet \overline{(A - B - C)}.$$

Now use DeMorgan's theorems to rewrite the last factor:

$$E = ((A - B) + (A - C) + (B - C)) - (\overline{A} + \overline{B} + \overline{C})$$

Now distribute the last factor:

$$E = ((A - B) \bullet (\overline{A} + \overline{B} + \overline{C})) + ((A - C) \bullet (\overline{A} + \overline{B} + \overline{C})) + (\{B \bullet C) - (\overline{A} + \overline{B} + \overline{C}))$$

Now distribute within each term; we show one example:

$$((A - B) \bullet (\overline{A} + \overline{B} + \overline{C})) = (A - B - \overline{A}) + (A'B \bullet \overline{B}) + (A'B - \overline{C}) = 0 + 0 + (A - B - \overline{C})$$

(This is simply $A \bullet B \bullet \overline{C}$.) Thus, the equation above becomes

$E = (A - B \bullet \overline{C}) + (A \bullet \overline{B} \bullet C) + (\overline{A} \bullet B - C)$, which is the desired result.

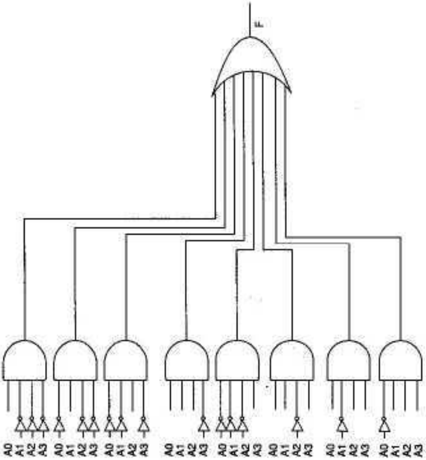
B.7 Four inputs A0-A3 & F (0/P) = 1 if an odd number of Is exist in A.

A3	A2	A1	A0	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	a	0	0
0	1	l	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

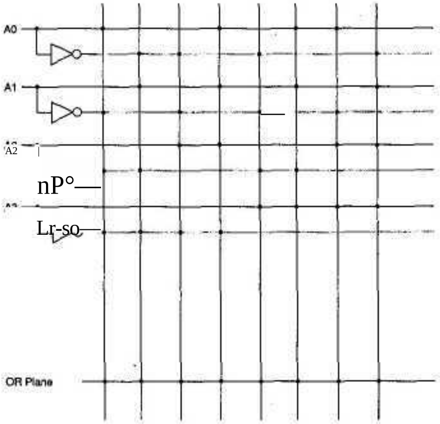
B.8 F = A3'A2'A1'AO + A3'A2'A1 AO' + A3'A2 Al'AO' + A3'A2 AI AO +

A3 A2'A1'AO' + A3 A2'A1 AO + A3'A2'A1 AO' + A3 A2 Al AO'

Note: F = AO XOR Al XOR A2 XOR A3. Another question can ask the students to prove that.


F
A0
A1
A2
A3
A0
A1
A2
A3
A0
A1
A2
A3
A0
A1
A2
A3
A0
A1
A2
A3
A0
A1
A2
A3
A0
A1
A2
A3
A0
A1
A2
A3

B.9



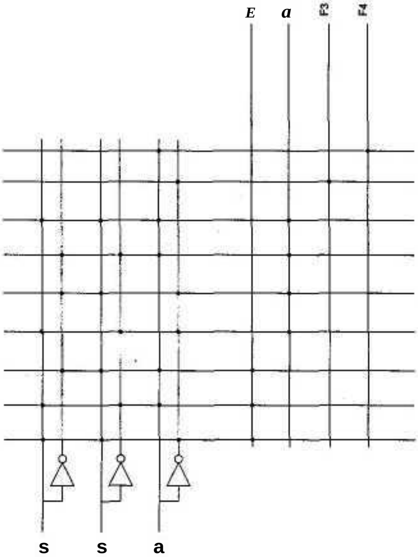
B.10 No solution provided.

B.11

x2	x1	x0	F1	F2	F3	F4
0	0	0	0	0	1	0
0	0	1	0	1	1	0
0	1	0	0	1	1	0
0	1	1	1	0	1	0
1	0	0	0	1	0	1
1	0	1	1	0	0	1
1	i	0	1	0	0	1
1	1	1	0	1	0	1

Fl = X2'XI XO + X2 Xl'XO + X2 XI XO'
F2 = X2TU-X0 + X2TU XO' + X2 Xl'XO' + X2 XI XO = (A XOR B XOR C)
F3=X2'
F4 = X2(=F3')

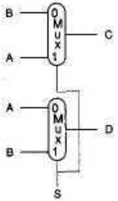
B.12



B.13

- $\overline{x_2}y_2 + x_2y_2\overline{x_1}y_1 + x_2y_2x_1y_1\overline{x_0}y_0 + \overline{x_2y_2x_1}y_1 + \overline{x_2y_2x_1y_1x_0}y_0 + \overline{x_2y_2}x_1y_1\overline{x_0}y_0 + x_2\overline{y_2x_1y_1}x_0y_0 + x_2y_2x_1y_1\overline{x_0}y_0$
- $x_2\overline{y_2} + x_2y_2\overline{x_1}y_1 + x_2y_2x_1y_1\overline{x_0}y_0 + \overline{x_2y_2x_1}y_1 + x_2\overline{y_2x_1y_1}x_0y_0 + \overline{x_2y_2x_1y_1x_0}y_0 + \overline{x_2y_2}x_1y_1\overline{x_0}y_0$
- $(x2_y2 + \overline{x2y2})(x_1y_1+x\sim ly_1)(x_0y_0 + \overline{x_0}y_1))$

B.14



B.15 Generalizing DeMorgan's theorems for this exercise, if $\overline{A + B} = \overline{A} \cdot \overline{B}$, then $\overline{A + B + C} = \overline{A+(B + C)} = \overline{A} \cdot \overline{(B + C)} = \overline{A} \cdot (\overline{B} \cdot \overline{C}) = \overline{A} \cdot \overline{B}\,\overline{C}$.

Similarly,

$\overline{A \cdot B \cdot C} = \overline{A \cdot (B \cdot C)} = \overline{A} + \overline{B \cdot C} = \overline{A} \cdot (\overline{B} + \overline{C}) = \overline{A} + \overline{B} + \overline{C}$.

Intuitively, DeMorgan's theorems say that (1) the negation of a sum-of-products form equals the product of the negated sums, and (2) the negation of a product-of-sums form equals the sum of the negated products. So,

$E = \overline{\overline{E}}$

$= \overline{\overline{(A \cdot B\overline{C}) + (A\,C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})}}$

$= \overline{\overline{(A \cdot B \cdot \overline{C})} \cdot \overline{(A \cdot C \cdot \overline{B})} \cdot \overline{(B \cdot C \cdot \overline{A})}}$; first application of DeMorgan's theorem

$= \overline{(\overline{A} + \overline{B} + C) \cdot (\overline{A} + \overline{C} + B) \cdot (\overline{B} + \overline{C} + A)}$; second application of DeMorgan's theorem and product-of-sums form

B.16 No solution provided.

B.18 2-1 multiplexor and 8 bit up/down counter.

B.19

```
module  LATCHCclock.D.Q.Qbar)
input  clock,D;
reg  0;
wire  Obar:
assign  Qbar  -  ~0;
always  @(D.clock)   //senstivity  list  watches  clock  and  data
begi n
   if(clock)
     0  -  D;
end
endmodule
```

B.20

```
    module  decoder (in, out,, enable);
    input [1:0] in;
    input enable
    output [3:0] out;
    reg [3:0] out;

    always @ (enable. in)
      if (enable) begin
      out = 0;
      case (in)
             2'hO : out - 4'hl;
             2'hl : out - 4'h2;
             2'h2 : out = 4'h4;
             2'h3 : out = 4'h8;
      endcase
    end
    endmodule
```

B.21

```
module ACCIClk, Rst. Load, IN, LOAD, OUT);

input C1k, Rst, Load;
input [3:0] IN;
input [15:0] LOAD
output [15:0] OUT;

wire [15:0] W;
reg [15:0] Register;

initial begin
Register = 0;
end
assign W = IN + OUT;

always @ {Rst,Load)
begin
if Rst begin
        Register = 0;
end

if Load begin
        Register = LOAD;
end
end

always @ (Clk)
begin
        Register <- W;
end

endmodule
```

B.22 We use Figure 3.5 to implement the multiplier. We add a control signal "load" to load the multiplicand and the multiplier. The load signal also initiates the multiplication. An output signal "done" indicates that simulation is done.

```
module MULKclk, load, Multiplicand. Multiplier, Product, done);
input elk, load;
input [31:0] Multiplicand, Multiplier;
output [63:0] Product;
output done;

         reg [63:0] A, Product;
         reg [31:0] B;
         reg [5:0] loop;
         reg done;

         initial begin
           done - 0; loop = 0;
         end

         always @(posedge elk) begin
           if (load && loop ™Q) begin
                  done <- 0;
                  Product <=0;
                  A <= Multiplicand;
                  B <=.Multiplier;
                  loop <= 32;
         end

         ifdoop > 0) begin
             if(B[0] =- 1)
                Product <- Product + A;

             A <- A << 1;
             B <- 6 » 1;
             1 oop <= loop -1;

             if(loop — 0)
               dorie <- 1;
           end

         end
         endmodule
```

B.23 We use Figure 3.10 for divider implementation, with additions similar to the ones listed above in the answer for Exercise B.22.

```
module DIVtclk, load. Divisor, Dividend, Quotient, Remainder, done);

input clk, 1oad;
input [31:0] Divisor;
input [63:0] Dividend:
output [31:0] Quotient;
input [31:0] Remainder;
output done;

reg [31:0] Quotient;      //Quotient
reg [63:0] D, R;         //Divisor,  Remainder
reg [6:0] loop;          //Loop counter
reg done;

initial begin
  done = 0; 1oop = 0;
end

assign Remainder - R[31:0];

always ©Cposedge elk) begin
  if (load SS loop —0) begin
    done <= 0;
    R <=Dividend;
    D <- Divisor << 32;
        Quotient <-0;
        loop <= 33;
end

iffloop > 0) begin
        if(R - D >- 0)
          begin
          Quotient <= (Ouotient << 1) + 1;
          R <- R - D;
          end
        else
          begin
          Quotient <= Quotient << 1;
        end

        D <- D » 1:
        loop <= loop - 1;
```

```
          ifdoop -~ 0)
            done <= 1:
    end

  end
  endmodule
```

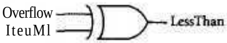
Note: This code does not check for division by zero (i.e., when Divisior - = 0) or for quotient overflow (i.e., when Divisior < = Dividiend [64:32]).

B.24 The ALU-supported set less than (s 11) uses just the sign bit. In this case, if we try a set less than operation using the values -7_{ten} and 6_{ten}, we would get -7 > 6. This is clearly wrong. Modify the 32-bit ALU in Figure 4.11 on page 169 to handle s 11 correctly by factor in overflow in the decision.

If there is no overflow, the calculation is done properly in Figure 4.17 and we simply use the sign bit (Result31). If there is overflow, however, then the sign bit is wrong and we need the inverse of the sign bit.

Overflow	Result31	LessThan
0	0	0
0	1	1
1	0	1
1	1	0

LessThan = Overflow © Result31



Overflow	Result31	LessThan
0	0	0
0	1	1
1	0	1
1	1	0

B.25 Given that a number that is greater than or equal to zero is termed positive and a number that is less than zero is negative, inspection reveals that the last two rows of Figure 4.44 restate the information of the first two rows. Because A - B = A + (-B), the operation A - B when A is positive and B negative is the same as the operation A + B when A is positive and B is positive. Thus the third row restates the conditions of the first. The second and fourth rows refer also to the same condition.

Because subtraction of two's complement numbers is performed by addition, a complete examination of overflow conditions for addition suffices to show also when overflow will occur for subtraction. Begin with the first two rows of Figure 4.44 and add rows for A and B with opposite signs. Build a table that shows all possible combinations of Sign and Carryin to the sign bit position and derive the CarryOut, Overflow, and related information. Thus,

Sign A	Sign B	Carry In	Carry Out	Sign of result	Correct sign of result	Over-flow?	Carry In XOR Carry Out	Notes
0	0	0	0	0	0	No	0	
0	0	1	0	1	0	*fes*	1	Carries differ
0	1	0	0	1	1	No	0	IAI < IBI
0	1	1	1	0	0	No	0	IAI > IBI
1	0	0	0	1	1	No	0	IAI > IBI
1	0	1	1	0	0	No	0	IAI < iBI
1	1	0	1	0	1	Yes	1	Carries differ
1	1	1	1	1	1	No	0	

From this table an Exclusive OR (XOR) of the Carryin and CarryOut of the sign bit serves to detect overflow. When the signs of A and B differ, the value of the Carryin is determined by the relative magnitudes of A and B, as listed in the Notes column.

B.26 Cl = C4, C2 = C8, C3 = Cl2, and C4 = cl6.

c4 = $G_{3,o}$ + (Pi,o • cO).

c8 is given in the exercise.

cl2 = G,,j, + (P_u,8 • $G_{,,4}$) + ($P_{n,8}$ • P,,4- G_M) + {Pn,»-P_7,4 • P3.0 • cO).

CIS = G_{15il2} + (P_{15},i_2 • G,,,,) + (P,5,$_{12}$ • $P_{u,8}$ • $G_{7|4}$)

+ (Pl5,12 ' Pll,« • P7.4 • $G_{3,0}$) + (Pl5,12 • Pll,» • P7.4 • ^{P}3.0 • CO).

B.27 The equations for c4, c8, and d2 are the same as those given in the solution to Exercise 4.44. Using 16-bit adders means using another level of carry lookahead logic to construct the 64-bit adder. The second level generate, G(f, and propagate, PO', are

$$GO' = G_{15,o} = Gis,_{12} + Pi5{,}12 \cdot G_{U,8} + P_{15},i2\text{-} P_{1\,U}\text{-}G7.4 + P_{15,12} \cdot P_{11j8} \cdot P_{7>4} \cdot G_{3l0}$$

and

$$PO' = P_{15,0} = P_{15,12} \cdot P_{11,8} \cdot P_{7,4} \cdot P_{3,0}$$

Using GO' and PO', we can write c16 more compactly as

$$c16 = G_{1SiO} + Pi_{5?o}\text{-}cO$$

and

$$c32 = G_{3U6} + P_3i_{i16} \cdot c16$$
$$c48 = G47_{i3}2 + P4_{7i}32\text{-}c32$$
$$c64 = G_63,_{4g} + P_63{,}4_8\text{-}c48$$

A 64-bit adder diagram in the style of Figure B.6.3 would look like the foUowing:

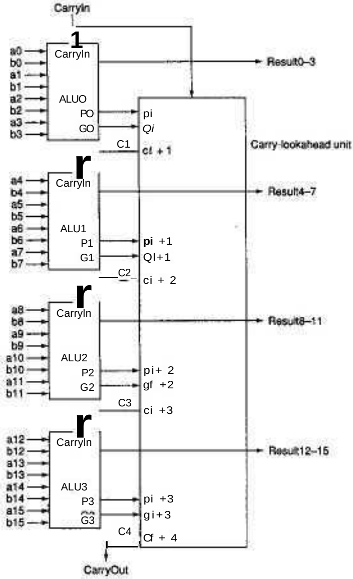


FIGURE B.8.3 Four 4-Ut ALUs u»b« carry lookahaad to form a 16-btt «dder. Note that the carries ccime from the carry-2ookahead unit, not from the 4-bit ALUs.

B.28 No solution provided.

B.29 No solution provided.

B.30 No solution provided.

B.31 No solution provided.

B.32 No solution provided.

B.33 No solution provided.

B.34 The longest paths through the top {ripple carry) adder organization in Figure B. 14.1 all start at input aO or bO and pass thrdiigh seven full adders on the way to output s4 or s5. There are many such paths, all with a time delay of 7 x 2T = 14T. The longest paths through the bottom (carry sale); adder all start at input bO, eO, fl), bl, el, or fl and proceed through six full adders to outputs s4 or s5. The time delay for this circuit is only 6 x 2T = 12T.