# 1

# Solutions

**1.1** **Personal computer (includes workstation and laptop):** Personal computers emphasize delivery of good performance to single users at low cost and usually execute third-party software.

**Personal mobile device (PMD, includes tablets):** PMDs are battery operated with wireless connectivity to the Internet and typically cost hundreds of dollars, and, like PCs, users can download software ("apps") to run on them. Unlike PCs, they no longer have a keyboard and mouse, and are more likely to rely on a touch-sensitive screen or even speech input.

**Server:** Computer used to run large problems and usually accessed via a network.

**Warehouse scale computer:** Thousands of processors forming a large cluster.

**Supercomputer:** Computer composed of hundreds to thousands of processors and terabytes of memory.

**Embedded computer:** Computer designed to run one application or one set of related applications and integrated into a single system.

**1.2**

**a.** Performance via Pipelining
**b.** Dependability via Redundancy
**c.** Performance via Prediction
**d.** Make the Common Case Fast
**e.** Hierarchy of Memories
**f.** Performance via Parallelism
**g.** Design for Moore's Law
**h.** Use Abstraction to Simplify Design

**1.3** The program is compiled into an assembly language program, which is then assembled into a machine language program.

**1.4**

**a.** 1280 × 1024 pixels = 1,310,720 pixels => 1,310,720 × 3 = 3,932,160 bytes/frame.

**b.** 3,932,160 bytes × (8 bits/byte) /100E6 bits/second = 0.31 seconds

**1.5**

**a.** performance of P1 (instructions/sec) = $3 \times 10^9/1.5 = 2 \times 10^9$
performance of P2 (instructions/sec) = $2.5 \times 10^9/1.0 = 2.5 \times 10^9$
performance of P3 (instructions/sec) = $4 \times 10^9/2.2 = 1.8 \times 10^9$

**b.** cycles(P1) = $10 \times 3 \times 10^9 = 30 \times 10^9$ s

cycles(P2) = $10 \times 2.5 \times 10^9 = 25 \times 10^9$ s

cycles(P3) = $10 \times 4 \times 10^9 = 40 \times 10^9$ s

**c.** No. instructions(P1) = $30 \times 10^9/1.5 = 20 \times 10^9$

No. instructions(P2) = $25 \times 10^9/1 = 25 \times 10^9$

No. instructions(P3) = $40 \times 10^9/2.2 = 18.18 \times 10^9$

$CPI_{new} = CPI_{old} \times 1.2$, then CPI(P1) = 1.8, CPI(P2) = 1.2, CPI(P3) = 2.6

$f$ = No. instr. × CPI/time, then

$f$(P1) = $20 \times 10^9 \times 1.8/7 = 5.14$ GHz

$f$(P2) = $25 \times 10^9 \times 1.2/7 = 4.28$ GHz

$f$(P1) = $18.18 \times 10^9 \times 2.6/7 = 6.75$ GHz

## 1.6

**a.** Class A: $10^5$ instr. Class B: $2 \times 10^5$ instr. Class C: $5 \times 10^5$ instr. Class D: $2 \times 10^5$ instr.

Time = No. instr. × CPI/clock rate

Total time P1 = $(10^5 + 2 \times 10^5 \times 2 + 5 \times 10^5 \times 3 + 2 \times 10^5 \times 3)/(2.5 \times 10^9) = 10.4 \times 10^{-4}$ s

Total time P2 = $(10^5 \times 2 + 2 \times 10^5 \times 2 + 5 \times 10^5 \times 2 + 2 \times 10^5 \times 2)/(3 \times 10^9) = 6.66 \times 10^{-4}$ s

CPI(P1) = $10.4 \times 10^{-4} \times 2.5 \times 10^9/10^6 = 2.6$

CPI(P2) = $6.66 \times 10^{-4} \times 3 \times 10^9/10^6 = 2.0$

**b.** clock cycles(P1) = $10^5 \times 1 + 2 \times 10^5 \times 2 + 5 \times 10^5 \times 3 + 2 \times 10^5 \times 3 = 26 \times 10^5$

clock cycles(P2) = $10^5 \times 2 + 2 \times 10^5 \times 2 + 5 \times 10^5 \times 2 + 2 \times 10^5 \times 2 = 20 \times 10^5$

## 1.7

**a.** CPI = $T_{exec} \times$ f/No. instr.

Compiler A CPI = 1.1

Compiler B CPI = 1.25

**b.** $f_B/f_A$ = (No. instr.(B) × CPI(B))/(No. instr.(A) × CPI(A)) = 1.37

**c.** $T_A/T_{new} = 1.67$

$T_B/T_{new} = 2.27$

## 1.8

**1.8.1** $C = 2 \times DP/(V^{2*}F)$

Pentium 4: $C = 3.2E{-}8F$

Core i5 Ivy Bridge: $C = 2.9E{-}8F$

**1.8.2** Pentium 4: $10/100 = 10\%$

Core i5 Ivy Bridge: $30/70 = 42.9\%$

**1.8.3** $(S_{new} + D_{new})/(S_{old} + D_{old}) = 0.90$

$D_{new} = C \times V_{new}\, 2 \times F$

$S_{old} = V_{old} \times I$

$S_{new} = V_{new} \times I$

Therefore:

$V_{new} = [D_{new}/(C \times F)]^{1/2}$

$D_{new} = 0.90 \times (S_{old} + D_{old}) - S_{new}$

$S_{new} = V_{new} \times (S_{old}/V_{old})$

Pentium 4:

$S_{new} = V_{new} \times (10/1.25) = V_{new} \times 8$

$D_{new} = 0.90 \times 100 - V_{new} \times 8 = 90 - V_{new} \times 8$

$V_{new} = [(90 - V_{new} \times 8)/(3.2E8 \times 3.6E9)]^{1/2}$

$V_{new} = 0.85\ V$

Core i5:

$S_{new} = V_{new} \times (30/0.9) = V_{new} \times 33.3$

$D_{new} = 0.90 \times 70 - V_{new} \times 33.3 = 63 - V_{new} \times 33.3$

$V_{new} = [(63 - V_{new} \times 33.3)/(2.9E8 \times 3.4E9)]^{1/2}$

$V_{new} = 0.64\ V$

## 1.9

**1.9.1**

| p | # arith inst. | # L/S inst. | # branch inst. | cycles | ex. time | speedup |
|---|---|---|---|---|---|---|
| 1 | 2.56E9 | 1.28E9 | 2.56E8 | 7.94E10 | 39.7 | 1 |
| 2 | 1.83E9 | 9.14E8 | 2.56E8 | 5.67E10 | 28.3 | 1.4 |
| 4 | 9.12E8 | 4.57E8 | 2.56E8 | 2.83E10 | 14.2 | 2.8 |
| 8 | 4.57E8 | 2.29E8 | 2.56E8 | 1.42E10 | 7.10 | 5.6 |

**1.9.2**

| p | ex. time |
|---|---|
| 1 | 41.0 |
| 2 | 29.3 |
| 4 | 14.6 |
| 8 | 7.33 |

**1.9.3** 3

## 1.10

**1.10.1** die area$_{15cm}$ = wafer area/dies per wafer = pi*7.5$^2$ / 84 = 2.10 cm$^2$

yield$_{15cm}$ = 1/(1+(0.020*2.10/2))$^2$ = 0.9593

die area$_{20cm}$ = wafer area/dies per wafer = pi*10$^2$/100 = 3.14 cm$^2$

yield$_{20cm}$ = 1/(1+(0.031*3.14/2))$^2$ = 0.9093

**1.10.2** cost/die$_{15cm}$ = 12/(84*0.9593) = 0.1489

cost/die$_{20cm}$ = 15/(100*0.9093) = 0.1650

**1.10.3** die area$_{15cm}$ = wafer area/dies per wafer = pi*7.5$^2$/(84*1.1) = 1.91 cm$^2$

yield$_{15cm}$ = 1/(1 + (0.020*1.15*1.91/2))$^2$ = 0.9575

die area$_{20cm}$ = wafer area/dies per wafer = pi*10$^2$/(100*1.1) = 2.86 cm$^2$

yield$_{20cm}$ = 1/(1 + (0.03*1.15*2.86/2))$^2$ = 0.9082

**1.10.4** defects per area$_{0.92}$ = (1–y^.5)/(y^.5*die_area/2) = (1−0.92^.5)/(0.92^.5*2/2) = 0.043 defects/cm$^2$

defects per area$_{0.95}$ = (1–y^.5)/(y^.5*die_area/2) = (1−0.95^.5)/(0.95^.5*2/2) = 0.026 defects/cm$^2$

## 1.11

**1.11.1** CPI = clock rate × CPU time/instr. count

clock rate = 1/cycle time = 3 GHz

CPI(bzip2) = 3 × 10$^9$ × 750/(2389 × 10$^9$)= 0.94

**1.11.2** SPEC ratio = ref. time/execution time

SPEC ratio(bzip2) = 9650/750 = 12.86

**1.11.3.** CPU time = No. instr. × CPI/clock rate

If CPI and clock rate do not change, the CPU time increase is equal to the increase in the of number of instructions, that is 10%.

**1.11.4** CPU time(before) = No. instr. × CPI/clock rate

CPU time(after) = 1.1 × No. instr. × 1.05 × CPI/clock rate

CPU time(after)/CPU time(before) = 1.1 × 1.05 =1.155. Thus, CPU time is increased by 15.5%.

**1.11.5** SPECratio = reference time/CPU time

SPECratio(after)/SPECratio(before) = CPU time(before)/CPU time(after) = 1/1.1555 = 0.86. The SPECratio is decreased by 14%.

**1.11.6** CPI = (CPU time × clock rate)/No. instr.

CPI = $700 \times 4 \times 10^9/(0.85 \times 2389 \times 10^9) = 1.37$

**1.11.7** Clock rate ratio = 4 GHz/3 GHz = 1.33

CPI @ 4 GHz = 1.37, CPI @ 3 GHz = 0.94, ratio = 1.45

They are different because, although the number of instructions has been reduced by 15%, the CPU time has been reduced by a lower percentage.

**1.11.8** 700/750 = 0.933. CPU time reduction: 6.7%

**1.11.9** No. instr. = CPU time × clock rate/CPI

No. instr. = $960 \times 0.9 \times 4 \times 10^9/1.61 = 2146 \times 10^9$

**1.11.10** Clock rate = No. instr. × CPI/CPU time.

Clock $\text{rate}_{\text{new}}$ = No. instr. × CPI/0.9 × CPU time = 1/0.9 clock $\text{rate}_{\text{old}}$ = 3.33 GHz

**1.11.11** Clock rate = No. instr. × CPI/CPU time.

Clock $\text{rate}_{\text{new}}$ = No. instr. × 0.85× CPI/0.80 CPU time = 0.85/0.80, clock $\text{rate}_{\text{old}}$ = 3.18 GHz

## 1.12

**1.12.1** $T(P1) = 5 \times 10^9 \times 0.9 / (4 \times 10^9) = 1.125$ s

$T(P2) = 10^9 \times 0.75 / (3 \times 10^9) = 0.25$ s

clock rate (P1) > clock rate(P2), performance(P1) < performance(P2)

**1.12.2** T(P1) = No. instr. × CPI/clock rate

T(P1) = 2.25 3 1021 s

T(P2) 5 N × 0.75/($3 \times 10^9$), then $N = 9 \times 10^8$

**1.12.3** MIPS = Clock rate × $10^{-6}$/CPI

MIPS(P1) = $4 \times 10^9 \times 10^{-6}/0.9 = 4.44 \times 10^3$

MIPS(P2) = $3 \times 10^9 \times 10^{-6}/0.75 = 4.0 \times 10^3$

MIPS(P1) > MIPS(P2), performance(P1) < performance(P2) (from 11a)

**1.12.4** MFLOPS = No. FP operations $\times 10^{-6}$/T

MFLOPS(P1) = .4 × 5E9 × 1E-6/1.125 = 1.78E3

MFLOPS(P2) = .4 × 1E9 × 1E-6/.25 = 1.60E3

MFLOPS(P1) > MFLOPS(P2), performance(P1) < performance(P2) (from 11a)

## 1.13

**1.13.1** $T_{fp} = 70 \times 0.8 = 56$ s. $T_{new} = 56+85+55+40 = 236$ s. Reduction: 5.6%

**1.13.2** $T_{new} = 250 \times 0.8 = 200$ s, $T_{fp}+T_{l/s}+T_{branch} = 165$ s, $T_{int} = 35$ s. Reduction time INT: 58.8%

**1.13.3** $T_{new} = 250 \times 0.8 = 200$ s, $T_{fp}+T_{int}+T_{l/s} = 210$ s. NO

## 1.14

**1.14.1** Clock cycles = $CPI_{fp} \times$ No. FP instr. + $CPI_{int} \times$ No. INT instr. + $CPI_{l/s} \times$ No. L/S instr. + $CPI_{branch} \times$ No. branch instr.

$T_{CPU}$ = clock cycles/clock rate = clock cycles/$2 \times 10^9$

clock cycles = $512 \times 10^6$; $T_{CPU} = 0.256$ s

To have the number of clock cycles by improving the CPI of FP instructions:

$CPI_{improved\ fp} \times$ No. FP instr. + $CPI_{int} \times$ No. INT instr. + $CPI_{l/s} \times$ No. L/S instr. + $CPI_{branch} \times$ No. branch instr. = clock cycles/2

$CPI_{improved\ fp}$ = (clock cycles/2 − ($CPI_{int} \times$ No. INT instr. + $CPI_{l/s} \times$ No. L/S instr. + $CPI_{branch} \times$ No. branch instr.)) / No. FP instr.

$CPI_{improved\ fp}$ = (256−462)/50 <0 ==> not possible

**1.14.2** Using the clock cycle data from a.

To have the number of clock cycles improving the CPI of L/S instructions:

$CPI_{fp} \times$ No. FP instr. + $CPI_{int} \times$ No. INT instr. + $CPI_{improved\ l/s} \times$ No. L/S instr. + $CPI_{branch} \times$ No. branch instr. = clock cycles/2

$CPI_{improved\ l/s}$ = (clock cycles/2 − ($CPI_{fp} \times$ No. FP instr. + $CPI_{int} \times$ No. INT instr. + $CPI_{branch} \times$ No. branch instr.)) / No. L/S instr.

$CPI_{improved\ l/s}$ = (256−198)/80 = 0.725

**1.14.3** Clock cycles = $CPI_{fp} \times$ No. FP instr. + $CPI_{int} \times$ No. INT instr. + $CPI_{l/s} \times$ No. L/S instr. + $CPI_{branch} \times$ No. branch instr.

$T_{CPU}$ = clock cycles/clock rate = clock cycles/$2 \times 10^9$

$CPI_{int} = 0.6 \times 1 = 0.6$; $CPI_{fp} = 0.6 \times 1 = 0.6$; $CPI_{l/s} = 0.7 \times 4 = 2.8$; $CPI_{branch} = 0.7 \times 2 = 1.4$

$T_{CPU}$ (before improv.) = 0.256 s; $T_{CPU}$ (after improv.)= 0.171 s

## 1.15

| processors | exec. time/ processor | time w/overhead | speedup | actual speedup/ideal speedup |
|---|---|---|---|---|
| 1 | 100 | | | |
| 2 | 50 | 54 | 100/54 = 1.85 | 1.85/2 = .93 |
| 4 | 25 | 29 | 100/29 = 3.44 | 3.44/4 = 0.86 |
| 8 | 12.5 | 16.5 | 100/16.5 = 6.06 | 6.06/8 = 0.75 |
| 16 | 6.25 | 10.25 | 100/10.25 = 9.76 | 9.76/16 = 0.61 |

# 2

# Solutions

**2.1**
```
addi f, h, -5 (note, no subi)
add  f, f, g
```

**2.2**
```
f = g + h + i
```

**2.3**
```
sub $t0, $s3, $s4
add $t0, $s6, $t0
lw  $t1, 16($t0)
sw  $t1, 32($s7)
```

**2.4**
```
B[g] = A[f] + A[1+f];
```

**2.5**
```
add $t0, $s6, $s0
add $t1, $s7, $s1
lw  $s0, 0($t0)
lw  $t0, 4($t0)
add $t0, $t0, $s0
sw  $t0, 0($t1)
```

**2.6**

**2.6.1**
```
temp = Array[0];
temp2 = Array[1];
Array[0] = Array[4];
Array[1] = temp;
Array[4] = Array[3];
Array[3] = temp2;
```

**2.6.2**
```
lw $t0, 0($s6)
lw $t1, 4($s6)
lw $t2, 16($s6)
sw $t2, 0($s6)
sw $t0, 4($s6)
lw $t0, 12($s6)
sw $t0, 16($s6)
sw $t1, 12($s6)
```

## 2.7

| Little-Endian | | Big-Endian | |
|---|---|---|---|
| Address | Data | Address | Data |
| 12 | ab | 12 | 12 |
| 8 | cd | 8 | ef |
| 4 | ef | 4 | cd |
| 0 | 12 | 0 | ab |

## 2.8 2882400018

## 2.9
```
sll   $t0, $s1, 2   # $t0 <-- 4*g
add   $t0, $t0, $s7 # $t0 <-- Addr(B[g])
lw    $t0, 0($t0)   # $t0 <-- B[g]
addi  $t0, $t0, 1   # $t0 <-- B[g]+1
sll   $t0, $t0, 2   # $t0 <-- 4*(B[g]+1) = Addr(A[B[g]+1])
lw    $s0, 0($t0)   # f    <-- A[B[g]+1]
```

## 2.10
```
f = 2*(&A);
```

## 2.11

| | type | opcode | rs | rt | rd | immed |
|---|---|---|---|---|---|---|
| addi $t0, $s6, 4 | I-type | 8 | 22 | 8 | | 4 |
| add $t1, $s6, $0 | R-type | 0 | 22 | 0 | 9 | |
| sw $t1, 0($t0) | I-type | 43 | 8 | 9 | | 0 |
| lw $t0, 0($t0) | I-type | 35 | 8 | 8 | | 0 |
| add $s0, $t1, $t0 | R-type | 0 | 9 | 8 | 16 | |

## 2.12

**2.12.1** 50000000

**2.12.2** overflow

**2.12.3** B0000000

**2.12.4** no overflow

**2.12.5** D0000000

**2.12.6** overflow

## 2.13

**2.13.1** $128 + \times > 2^{31}-1$, $x > 2^{31}-129$ and $128 + x < -2^{31}$, $x < -2^{31} - 128$ (impossible)

**2.13.2** $128 - x > 2^{31}-1$, $x < -2^{31}+129$ and $128 - x < -2^{31}$, $x > 2^{31} + 128$ (impossible)

**2.13.3** $x - 128 < -2^{31}$, $x < -2^{31} + 128$ and $x - 128 > 2^{31} - 1$, $x > 2^{31} + 127$ (impossible)

**2.14** r-type, add $s0, $s0, $s0

**2.15** i-type, 0xAD490020

**2.16** r-type, sub $v1, $v1, $v0, 0x00621822

**2.17** i-type, lw $v0, 4($at), 0x8C220004

**2.18**

**2.18.1** opcode would be 8 bits, rs, rt, rd fields would be 7 bits each

**2.18.2** opcode would be 8 bits, rs and rt fields would be 7 bits each

**2.18.3** more registers → more bits per instruction → could increase code size
more registers → less register spills → less instructions
more instructions → more appropriate instruction → decrease code size
more instructions → larger opcodes → larger code size

**2.19**

**2.19.1** 0xBABEFEF8

**2.19.2** 0xAAAAAAA0

**2.19.3** 0x00005545

**2.20**
```
srl $t0, $t0, 11
sll $t0, $t0, 26
ori $t2, $0, 0x03ff
sll $t2, $t2, 16
ori $t2, $t2, 0xffff
and $t1, $t1, $t2
or  $t1, $t1, $t0
```

**2.21**
```
nor $t1, $t2, $t2
```

**2.22**
```
lw  $t3, 0($s1)
sll $t1, $t3, 4
```

**2.23** $t2 = 3

**2.24** jump: no, beq: no

## 2.25

**2.25.1** i-type

**2.25.2**
```
addi $t2, $t2, –1
beq  $t2, $0, loop
```

## 2.26

**2.26.1** 20

**2.26.2**
```
i = 10;
do {
   B += 2;
   i = i – 1;
} while ( i > 0)
```

**2.26.3** 5*N

**2.27**
```
       addi $t0, $0, 0
       beq  $0,  $0, TEST1
LOOP1: addi $t1, $0, 0
       beq  $0,  $0, TEST2
LOOP2: add  $t3, $t0, $t1
       sll  $t2, $t1, 4
       add  $t2, $t2, $s2
       sw   $t3, ($t2)
       addi $t1, $t1, 1
TEST2: slt  $t2, $t1, $s1
       bne  $t2, $0, LOOP2
       addi $t0, $t0, 1
TEST1: slt  $t2, $t0, $s0
       bne  $t2, $0, LOOP1
```

**2.28** 14 instructions to implement and 158 instructions executed

**2.29**
```
for (i=0; i<100; i++) {
    result += MemArray[s0];
    s0 = s0 + 4;
}
```

**2.30**
```
      addi $t1, $s0, 400
LOOP: lw   $s1, 0($t1)
      add  $s2, $s2, $s1
      addi $t1, $t1, -4
      bne  $t1, $s0, LOOP
```

**2.31**
```
fib:   addi $sp, $sp, -12       # make room on stack
       sw   $ra, 8($sp)         # push $ra
       sw   $s0, 4($sp)         # push $s0
       sw   $a0, 0($sp)         # push $a0 (N)
       bgt  $a0, $0, test2      # if n>0, test if n=1
       add  $v0, $0, $0         # else fib(0) = 0
       j rtn                    #
test2: addi $t0, $0, 1          #
       bne  $t0, $a0, gen       # if n>1, gen
       add  $v0, $0, $t0        # else fib(1) = 1
       j rtn
gen:   subi $a0, $a0,1          # n-1
       jal  fib                 # call fib(n-1)
       add  $s0, $v0, $0        # copy fib(n-1)
       sub  $a0, $a0,1          # n-2
       jal  fib                 # call fib(n-2)
       add  $v0, $v0, $s0       # fib(n-1)+fib(n-2)
rtn:   lw   $a0, 0($sp)         # pop $a0
       lw   $s0, 4($sp)         # pop $s0
       lw   $ra, 8($sp)         # pop $ra
       addi $sp, $sp, 12        # restore sp
       jr   $ra

# fib(0) = 12 instructions, fib(1) = 14 instructions,
# fib(N) = 26 + 18N instructions for N >=2
```

**2.32** Due to the recursive nature of the code, it is not possible for the compiler to in-line the function call.

**2.33**
```
after calling function fib:
old $sp ->  0x7fffffc   ???
            -4          contents of register $ra for
                        fib(N)
            -8          contents of register $s0 for
                        fib(N)
$sp->       -12         contents of register $a0 for
                        fib(N)
there will be N-1 copies of $ra, $s0 and $a0
```

**2.34**
```
f: addi $sp,$sp,-12
   sw     $ra,8($sp)
   sw     $s1,4($sp)
   sw     $s0,0($sp)
   move   $s1,$a2
   move   $s0,$a3
   jal    func
   move   $a0,$v0
   add    $a1,$s0,$s1
   jal    func
   lw     $ra,8($sp)
   lw     $s1,4($sp)
   lw     $s0,0($sp)
   addi   $sp,$sp,12
   jr     $ra
```

**2.35** We can use the tail-call optimization for the second call to `func`, but then we must restore `$ra`, `$s0`, `$s1`, and `$sp` before that call. We save only one instruction (`jr $ra`).

**2.36** Register `$ra` is equal to the return address in the caller function, registers `$sp` and `$s3` have the same values they had when function `f` was called, and register `$t5` can have an arbitrary value. For register `$t5`, note that although our function `f` does not modify it, function `func` is allowed to modify it so we cannot assume anything about the of `$t5` after function `func` has been called.

**2.37**
```
MAIN:  addi $sp, $sp, -4
       sw   $ra, ($sp)
       add  $t6, $0, 0x30 # '0'
       add  $t7, $0, 0x39 # '9'
       add  $s0, $0, $0
       add  $t0, $a0, $0
LOOP:  lb   $t1, ($t0)
       slt  $t2, $t1, $t6
       bne  $t2, $0, DONE
       slt  $t2, $t7, $t1
       bne  $t2, $0, DONE
       sub  $t1, $t1, $t6
       beq  $s0, $0, FIRST
       mul  $s0, $s0, 10
FIRST: add  $s0, $s0, $t1
       addi $t0, $t0, 1
       j LOOP
```

```
DONE:  add  $v0, $s0, $0
       lw   $ra, ($sp)
       addi $sp, $sp, 4
       jr   $ra
```

**2.38** 0x00000011

**2.39** Generally, all solutions are similar:

```
lui $t1, top_16_bits
ori $t1, $t1, bottom_16_bits
```

**2.40** No, jump can go up to 0x0FFFFFFC.

**2.41** No, range is 0x604 + 0x1FFFC = 0x0002 0600 to 0x604 - 0x20000 = 0xFFFE 0604.

**2.42** Yes, range is 0x1FFFF004 + 0x1FFFC = 0x2001F000 to 0x1FFFF004 - 0x20000 = 1FFDF004

**2.43**
```
trylk: li   $t1,1
       ll   $t0,0($a0)
       bnez $t0,trylk
       sc   $t1,0($a0)
       beqz $t1,trylk
       lw   $t2,0($a1)
       slt  $t3,$t2,$a2
       bnez $t3,skip
       sw   $a2,0($a1)
skip:  sw   $0,0($a0)
```

**2.44**
```
try: ll   $t0,0($a1)
     slt  $t1,$t0,$a2
     bnez $t1,skip
     mov  $t0,$a2
     sc   $t0,0($a1)
     beqz $t0,try
skip:
```

**2.45** It is possible for one or both processors to complete this code without ever reaching the SC instruction. If only one executes SC, it completes successfully. If both reach SC, they do so in the same cycle, but one SC completes first and then the other detects this and fails.

## 2.46

**2.46.1** Answer is no in all cases. Slows down the computer.

CCT = clock cycle time

ICa = instruction count (arithmetic)

ICls = instruction count (load/store)

ICb = instruction count (branch)

new CPU time = 0.75*old ICa*CPIa*1.1*oldCCT

+ oldICls*CPIls*1.1*oldCCT

+ oldICb*CPIb*1.1*oldCCT

The extra clock cycle time adds sufficiently to the new CPU time such that it is not quicker than the old execution time in all cases.

**2.46.2** 107.04%, 113.43%

## 2.47

**2.47.1** 2.6

**2.47.2** 0.88

**2.47.3** 0.533333333

# 3

# Solutions

**3.1** 5730

**3.2** 5730

**3.3** 0101111011010100

The attraction is that each hex digit contains one of 16 different characters (0–9, A–E). Since with 4 binary bits you can represent 16 different patterns, in hex each digit requires exactly 4 binary bits. And bytes are by definition 8 bits long, so two hex digits are all that are required to represent the contents of 1 byte.

**3.4** 753

**3.5** 7777 (−3777)

**3.6** Neither (63)

**3.7** Neither (65)

**3.8** Overflow (result = −179, which does not fit into an SM 8-bit format)

**3.9** −105 − 42 = −128 (−147)

**3.10** −105 + 42 = −63

**3.11** 151 + 214 = 255 (365)

**3.12** 62×12

| Step | Action | Multiplier | Multiplicand | Product |
|---|---|---|---|---|
| 0 | Initial Vals | 001 010 | 000 000 110 010 | 000 000 000 000 |
| 1 | lsb=0, no op | 001 010 | 000 000 110 010 | 000 000 000 000 |
| | Lshift Mcand | 001 010 | 000 001 100 100 | 000 000 000 000 |
| | Rshift Mplier | 000 101 | 000 001 100 100 | 000 000 000 000 |
| 2 | Prod=Prod+Mcand | 000 101 | 000 001 100 100 | 000 001 100 100 |
| | Lshift Mcand | 000 101 | 000 011 001 000 | 000 001 100 100 |
| | Rshift Mplier | 000 010 | 000 011 001 000 | 000 001 100 100 |
| 3 | lsb=0, no op | 000 010 | 000 011 001 000 | 000 001 100 100 |
| | Lshift Mcand | 000 010 | 000 110 010 000 | 000 001 100 100 |
| | Rshift Mplier | 000 001 | 000 110 010 000 | 000 001 100 100 |
| 4 | Prod=Prod+Mcand | 000 001 | 000 110 010 000 | 000 111 110 100 |
| | Lshift Mcand | 000 001 | 001 100 100 000 | 000 111 110 100 |
| | Rshift Mplier | 000 000 | 001 100 100 000 | 000 111 110 100 |
| 5 | lsb=0, no op | 000 000 | 001 100 100 000 | 000 111 110 100 |
| | Lshift Mcand | 000 000 | 011 001 000 000 | 000 111 110 100 |
| | Rshift Mplier | 000 000 | 011 001 000 000 | 000 111 110 100 |
| 6 | lsb=0, no op | 000 000 | 110 010 000 000 | 000 111 110 100 |
| | Lshift Mcand | 000 000 | 110 010 000 000 | 000 111 110 100 |
| | Rshift Mplier | 000 000 | 110 010 000 000 | 000 111 110 100 |

**3.13** 62×12

| Step | Action | Multiplicand | Product/Multiplier |
|---|---|---|---|
| 0 | Initial Vals | 110 010 | 000 000 001 010 |
| 1 | lsb=0, no op | 110 010 | 000 000 001 010 |
| | Rshift Product | 110 010 | 000 000 000 101 |
| 2 | Prod=Prod+Mcand | 110 010 | 110 010 000 101 |
| | Rshift Mplier | 110 010 | 011 001 000 010 |
| 3 | lsb=0, no op | 110 010 | 011 001 000 010 |
| | Rshift Mplier | 110 010 | 001 100 100 001 |
| 4 | Prod=Prod+Mcand | 110 010 | 111 110 100 001 |
| | Rshift Mplier | 110 010 | 011 111 010 000 |
| 5 | lsb=0, no op | 110 010 | 011 111 010 000 |
| | Rshift Mplier | 110 010 | 001 111 101 000 |
| 6 | lsb=0, no op | 110 010 | 001 111 101 000 |
| | Rshift Mplier | 110 010 | 000 111 110 100 |

**3.14** For hardware, it takes 1 cycle to do the add, 1 cycle to do the shift, and 1 cycle to decide if we are done. So the loop takes (3 × A) cycles, with each cycle being B time units long.

For a software implementation, it takes 1 cycle to decide what to add, 1 cycle to do the add, 1 cycle to do each shift, and 1 cycle to decide if we are done. So the loop takes (5 × A) cycles, with each cycle being B time units long.

(3×8)×4tu = 96 time units for hardware

(5×8)×4tu = 160 time units for software

**3.15** It takes B time units to get through an adder, and there will be A − 1 adders. Word is 8 bits wide, requiring 7 adders. 7×4tu = 28 time units.

**3.16** It takes B time units to get through an adder, and the adders are arranged in a tree structure. It will require log2(A) levels. 8 bit wide word requires 7 adders in 3 levels. 3×4tu = 12 time units.

**3.17** 0x33 × 0x55 = 0x10EF. 0x33 = 51, and 51 = 32+16+2+1. We can shift 0x55 left 5 places (0xAA0), then add 0x55 shifted left 4 places (0x550), then add 0x55 shifted left once (0xAA), then add 0x55. 0xAA0+0x550+0xAA+0x55 = 0x10EF. 3 shifts, 3 adds.

(Could also use 0x55, which is 64+16+4+1, and shift 0x33 left 6 times, add to it 0x33 shifted left 4 times, add to that 0x33 shifted left 2 times, and add to that 0x33. Same number of shifts and adds.)

**3.18** 74/21 = 3 remainder 9

| Step | Action | Quotient | Divisor | Remainder |
|---|---|---|---|---|
| 0 | Initial Vals | 000 000 | 010 001 000 000 | 000 000 111 100 |
| 1 | Rem=Rem–Div | 000 000 | 010 001 000 000 | 101 111 111 100 |
| | Rem<0,R+D,Q<< | 000 000 | 010 001 000 000 | 000 000 111 100 |
| | Rshift Div | 000 000 | 001 000 100 000 | 000 000 111 100 |
| 2 | Rem=Rem–Div | 000 000 | 001 000 100 000 | 111 000 011 100 |
| | Rem<0,R+D,Q<< | 000 000 | 001 000 100 000 | 000 000 111 100 |
| | Rshift Div | 000 000 | 000 100 010 000 | 000 000 111 100 |
| 3 | Rem=Rem–Div | 000 000 | 000 100 010 000 | 111 100 101 100 |
| | Rem<0,R+D,Q<< | 000 000 | 000 100 010 000 | 000 000 111 100 |
| | Rshift Div | 000 000 | 000 010 001 000 | 000 000 111 100 |
| 4 | Rem=Rem–Div | 000 000 | 000 010 001 000 | 111 110 110 100 |
| | Rem<0,R+D,Q<< | 000 000 | 000 010 001 000 | 000 000 111 100 |
| | Rshift Div | 000 000 | 000 001 000 100 | 000 000 111 100 |
| 5 | Rem=Rem–Div | 000 000 | 000 001 000 100 | 111 111 111 000 |
| | Rem<0,R+D,Q<< | 000 000 | 000 001 000 100 | 000 000 111 100 |
| | Rshift Div | 000 000 | 000 000 100 010 | 000 000 111 100 |
| 6 | Rem=Rem–Div | 000 000 | 000 000 100 010 | 000 000 011 010 |
| | Rem>0,Q<<1 | 000 001 | 000 000 100 010 | 000 000 011 010 |
| | Rshift Div | 000 001 | 000 000 010 001 | 000 000 011 010 |
| 7 | Rem=Rem–Div | 000 001 | 000 000 010 001 | 000 000 001 001 |
| | Rem>0,Q<<1 | 000 011 | 000 000 010 001 | 000 000 001 001 |
| | Rshift Div | 000 011 | 000 000 001 000 | 000 000 001 001 |

**3.19.** In these solutions a 1 or a 0 was added to the Quotient if the remainder was greater than or equal to 0. However, an equally valid solution is to shift in a 1 or 0, but if you do this you must do a compensating right shift of the remainder (only the remainder, not the entire remainder/quotient combination) after the last step.

74/21 = 3 remainder 11

| Step | Action | Divisor | Remainder/Quotient |
|---|---|---|---|
| 0 | Initial Vals | 010 001 | 000 000 111 100 |
| 1 | R<< | 010 001 | 000 001 111 000 |
| | Rem=Rem–Div | 010 001 | 111 000 111 000 |
| | Rem<0,R+D | 010 001 | 000 001 111 000 |
| 2 | R<< | 010 001 | 000 011 110 000 |
| | Rem=Rem–Div | 010 001 | 110 010 110 000 |
| | Rem<0,R+D | 010 001 | 000 011 110 000 |
| 3 | R<< | 010 001 | 000 111 100 000 |
| | Rem=Rem–Div | 010 001 | 110 110 110 000 |
| | Rem<0,R+D | 010 001 | 000 111 100 000 |
| 4 | R<< | 010 001 | 001 111 000 000 |
| | Rem=Rem–Div | 010 001 | 111 110 000 000 |
| | Rem<0,R+D | 010 001 | 001 111 000 000 |

| Step | Action | Divisor | Remainder/Quotient |
|---|---|---|---|
| 5 | R<< | 010 001 | 011 110 000 000 |
| | Rem=Rem–Div | 010 001 | 111 110 000 000 |
| | Rem>0,R0=1 | 010 001 | 001 101 000 001 |
| 6 | R<< | 010 001 | 011 010 000 010 |
| | Rem=Rem–Div | 010 001 | 001 001 000 010 |
| | Rem>0,R0=1 | 010 001 | 001 001 000 011 |

**3.20** `201326592` in both cases.

**3.21** `jal 0x00000000`

**3.22**

0×0C000000 = 0000 1100 0000 0000 0000 0000 0000 0000

= 0 0001 1000 0000 0000 0000 0000 0000 000

sign is positive

exp = 0×18 = 24 − 127 = −103

there is a hidden 1

mantissa = 0

answer = $1.0 \times 2^{-103}$

**3.23** $63.25 \times 10^0 = 111111.01 \times 2^0$

normalize, move binary point 5 to the left

$1.1111101 \times 2^5$

sign = positive, exp = 127+5=132

Final bit pattern: 0 1000 0100 1111 1010 0000 0000 0000 000

= 0100 0010 0111 1101 0000 0000 0000 0000 = 0x427D0000

**3.24** $63.25 \times 10^0 = 111111.01 \times 2^0$

normalize, move binary point 5 to the left

$1.1111101 \times 2^5$

sign = positive, exp = 1023+5=1028

Final bit pattern:

0 100 0000 0100 1111 1010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
= 0x404FA00000000000

**3.25** $63.25 \times 10^0 = 111111.01 \times 2^0 = 3F.40 \times 16^0$

move hex point 2 to the left

$.3F40 \times 16^2$

sign = positive, exp = 64+2

Final bit pattern: 01000010001111110100000000000000

**3.26** $-1.5625 \times 10^{-1} = -.15625 \times 10^0$

$= -.00101 \times 2^0$

move the binary point 2 to the right

$= -.101 \times 2^{-2}$

exponent = −2, fraction = −.10100000000000000000000

answer: 11111111111010110000000000000000000

**3.27** $-1.5625 \times 10^{-1} = -.15625 \times 10^0$

$= -.00101 \times 2^0$

move the binary point 3 to the right, $= -1.01 \times 2^{-3}$

exponent = −3 = −3+15 = 12, fraction = −.0100000000

answer: 1011000100000000

**3.28** $-1.5625 \times 10^{-1} = -.15625 \times 10^0$

$= -.00101 \times 2^0$

move the binary point 2 to the right

$= -.101 \times 2^{-2}$

exponent = −2, fraction = −.10100000000000000000000000000

answer: 10110000000000000000000000000101

**3.29** $2.6125 \times 10^1 + 4.150390625 \times 10^{-1}$

$2.6125 \times 10^1 = 26.125 = 11010.001 = 1.1010001000 \times 2^4$

$4.150390625 \times 10^{-1} = .4150390625 = .011010100111 = 1.1010100111 \times 2^{-2}$

Shift binary point 6 to the left to align exponents,

```
              GR
1.1010001000 00
1.0000011010 10 0111 (Guard 5 1, Round 5 0,
Sticky 5 1)
--------------------
1.1010100010 10
```

In this case the extra bit (G,R,S) is more than half of the least significant bit (0).

Thus, the value is rounded up.

$1.1010100011 \times 2^4 = 11010.100011 \times 2^0 = 26.546875 = 2.6546875 \times 10^1$

**3.30** $-8.0546875 \times -1.79931640625 \times 10^{-1}$

$-8.0546875 = -1.0000000111 \times 2^3$

$-1.79931640625 \times 10^{-1} = -1.0111000010 \times 2^{-3}$

Exp: $-3 + 3 = 0$, $0+16 = 16$ (10000)

Signs: both negative, result positive

Fraction:

```
              1.0000000111
            × 1.0111000010
              ------------
              00000000000
             10000000111
            00000000000
           00000000000
          00000000000
         00000000000
        10000000111
       10000000111
      10000000111
     00000000000
    10000000111
1.0111001100001001110
```

1.0111001100 00 01001110 Guard = 0, Round = 0, Sticky = 1:NoRnd

$1.0111001100 \times 2^0 = 0100000111001100$ ($1.0111001100 = 1.44921875$)

$-8.0546875 \times -.179931640625 = 1.4492931365966796875$

Some information was lost because the result did not fit into the available 10-bit field. Answer (only) off by .0000743865966796875

**3.31** $8.625 \times 10^1 / -4.875 \times 10^0$

$8.625 \times 10^1 = 1.0101100100 \times 2^6$

$-4.875 = -1.0011100000 \times 2^2$

Exponent = 6−2 = 4, 4+15 = 19 (10011)

Signs: one positive, one negative, result negative

Fraction:

```
                            1.00011011000100111
10011100000. | 10101100100.0000000000000000
             −10011100000.
               --------------
                10000100.0000
               −1001110.0000
                -------------
                 1100110.00000
                −100111.00000
                 --------------
                    1111.0000000
                  −1001.1100000
                   -------------
                     101.01000000
                   −100.11100000
                    -------------
                     000.011000000000
                       −.010011100000
                        --------------
                         .000100100000000
                        −.000010011100000
                         -----------------
                          .0000100001000000
                        −.0000010011100000
                         ------------------
                          .00000011011000000
                        −.00000010011100000
                         --------------------
                          .00000000110000000
```

1.000110110001001111 Guard=0, Round=1, Sticky=1: No Round, fix sign

$-1.0001101100 \times 2^4 = 1101000001101100 = 10001.101100 = -17.6875$

$86.25 / -4.875 = -17.692307692307$

Some information was lost because the result did not fit into the available 10-bit field. Answer off by .00480769230

**3.32** $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^3)$

$3.984375 \times 10^{-1} = 1.1001100000 \times 2^{-2}$

$3.4375 \times 10^{-1} = 1.0110000000 \times 2^{-2}$

$1.771 \times 10^3 = 1771 = 1.1011101011 \times 2^{10}$

shift binary point of smaller left 12 so exponents match

```
(A)         1.1001100000
(B)       + 1.0110000000
           -------------
           10.1111100000 Normalize,
(A+B)       1.0111110000 × 2^-1
(C)       + 1.1011101011
(A+B)         .0000000000 10 111110000 Guard = 1,
                          Round = 0, Sticky = 1
          ---------------
(A+B)+C   + 1.1011101011 10 1 Round up
```

(A+B)+C =$1.1011101100 \times 2^{10}$ = 0110101011101100 = 1772

**3.33** $3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^3)$

$3.984375 \times 10^{-1} = 1.1001100000 \times 2^{-2}$

$3.4375 \times 10^{-1} = 1.0110000000 \times 2^{-2}$

$1.771 \times 10^3 = 1771 = 1.1011101011 \times 2^{10}$

shift binary point of smaller left 12 so exponents match

```
(B)           .0000000000 01 0110000000 Guard = 0,
                          Round = 1, Sticky = 1
(C)         + 1.1011101011
             -------------
(B+C)       + 1.1011101011
```

```
(A)        .0000000000  011001100000
         --------------
A+(B+C) +1.1011101011  No round
A+(B+C) +1.1011101011  × 2^10 = 0110101011101011 = 1771
```

**3.34** No, they are not equal: (A+B)+C = 1772, A+(B+C) = 1771 (steps shown above).

Exact: .398437 + .34375 + 1771 = 1771.742187

**3.35** $(3.41796875 \times 10^{-3} \times 6.34765625 \times 10^{-3}) \times 1.05625 \times 10^{2}$

(A) $3.41796875 \times 10^{-3} = 1.1100000000 \times 2^{-9}$

(B) $4.150390625 \times 10^{-3} = 1.0001000000 \times 2^{-8}$

(C) $1.05625 \times 10^{2} = 1.1010011010 \times 2^{6}$

Exp: $-9-8 = -17$

Signs: both positive, result positive

Fraction:

```
(A)                    1.1100000000
(B)                  × 1.0001000000
                       --------------
                   11100000000
                11100000000
               ----------------------
             1.11011100000000000000
A×B          1.1101110000 00 00000000
Guard = 0, Round = 0, Sticky = 0: No Round
```

A×B $1.1101110000 \times 2^{-17}$ UNDERFLOW: Cannot represent number

**3.36** $3.41796875 \times 10^{-3} \times (6.34765625 \times 10^{-3} \times 1.05625 \times 10^{2})$

(A) $3.41796875 \times 10^{-3} = 1.1100000000 \times 2^{-9}$

(B) $4.150390625 \times 10^{-3} = 1.0001000000 \times 2^{-8}$

(C) $1.05625 \times 10^{2} = 1.1010011010 \times 2^{6}$

Exp: $-8+6 = -2$

Signs: both positive, result positive

Fraction:

```
(B)     1.0001000000
(C)   × 1.1010011010
          -------------
           10001000000
         10001000000
        10001000000
     10001000000
  10001000000
 10001000000
-----------------------
1.11000000111010000000 0
1.1100000011 10 100000000 Guard 5 1, Round 5 0, Sticky
5 1: Round
```

B×C $1.1100000100 \times 2^{-2}$

Exp: $-9-2 = -11$

Signs: both positive, result positive

Fraction:

```
(A)     1.1100000000
(B x C)× 1.1100000100
          -------------
           11100000000
     11100000000
    11100000000
   11100000000
 ----------------------
11.00010001110000000000     Normalize, add 1 to exponent
 1.1000100011 10 0000000000 Guard=1, Round=0, Sticky=0:
Round to even
```

A×(B×C) $1.1000100100 \times 2^{-10}$

**3.37** b) No:

$A \times B = 1.1101110000 \times 2^{-17}$ UNDERFLOW: Cannot represent

$A \times (B \times C) = 1.1000100100 \times 2^{-10}$

A and B are both small, so their product does not fit into the 16-bit floating point format being used.

**3.38** $1.666015625 \times 10^0 \times (1.9760 \times 10^4 - 1.9744 \times 10^4)$

(A) $1.666015625 \times 10^0 = 1.1010101010 \times 2^0$

(B) $1.9760 \times 10^4 = 1.0011010011 \times 2^{14}$

(C) $-1.9744 \times 10^4 = -1.0011010010 \times 2^{14}$

Exponents match, no shifting necessary

```
(B)      1.0011010011
(C)     −1.0011010010
         ----------------
(B+C)    0.0000000001 × 2^14
(B+C)    1.0000000000 × 2^4
```

Exp: 0+4 = 4

Signs: both positive, result positive

Fraction:

```
(A)                    1.1010101010
(B+C)                × 1.0000000000
                       ------------
             11010101010
            ------------------------
            1.10101010100000000000
A×(B+C)     1.1010101010 0000000000 Guard = 0, Round =
            0, sticky = 0: No round
```

$A \times (B+C)$ $1.1010101010 \times 2^4$

**3.39** $1.666015625 \times 10^0 \times (1.9760 \times 10^4 - 1.9744 \times 10^4)$

(A) $1.666015625 \times 10^0 = 1.1010101010 \times 2^0$

(B) $1.9760 \times 10^4 = 1.0011010011 \times 2^{14}$

(C) $-1.9744 \times 10^4 = -1.0011010010 \times 2^{14}$

Exp: 0+14 = 14

Signs: both positive, result positive

Fraction:

```
(A)                1.1010101010
(B)              × 1.0011010011
                   ------------
                    11010101010
                   11010101010
                  11010101010
                 11010101010
                11010101010
               11010101010
              ------------------------------
              10.0000001001100001111 Normalize, add 1 to
                                       exponent
A×B     1.0000000100 11 00001111 Guard = 1, Round = 1,
                                    Sticky = 1: Round
A×B     1.0000000101 × 2^15
```

Exp: 0+14=14

Signs: one negative, one positive, result negative

Fraction:

```
(A)                               1.1010101010
(C)                             × 1.0011010010
                                  ------------
                                  11010101010
                                 11010101010
                                11010101010
                               11010101010
                              11010101010
                             -------------------------
                              10.0000000111110111010
Normalize, add 1 to exponent
A×C                 1.0000000011 11 101110100
Guard = 1, Round = 1, Sticky = 1: Round
A×C       −1.0000000100 × 2^15
A×B             1.0000000101 × 2^15
A×C           −1.0000000100 × 2^15
                ---------------
A×B+A×C          .0000000001 × 2^15
A×B+A×C          1.0000000000 × 2^5
```

**3.40** b) No:

$A\times(B+C) = 1.1010101010 \times 2^4 = 26.65625$, and $(A\times B)+(A\times C) = 1.0000000000 \times 2^5 = 32$

Exact: $1.666015625 \times (19{,}760 - 19{,}744) = 26.65625$

**3.41**

| Answer | sign | exp | Exact? |
|---|---|---|---|
| 1 01111101 00000000000000000000000 | – | −2 | Yes |

**3.42** b+b+b+b = −1

b×4 = −1

They are the same

**3.43** 0101 0101 0101 0101 0101 0101

No

**3.44** 0011 0011 0011 0011 0011 0011

No

**3.45** 0101 0000 0000 0000 0000 0000

0.5

Yes

**3.46** 01010 00000 00000 00000

0.A

Yes

**3.47** Instruction assumptions:

(1) 8-lane 16-bit multiplies

(2) sum reductions of the four most significant 16-bit values

(3) shift and bitwise operations

(4) 128-, 64-, and 32-bit loads and stores of most significant bits

Outline of solution:

```
load register F[bits 127:0] = f[3..0] & f[3..0] (64-bit
load)
load register A[bits 127:0] = sig_in[7..0] (128-bit load)
```

```
 for i = 0 to 15 do
       load register B[bits 127:0] = sig_in[(i*8+7..i*8]
       (128-bit load)

       for j = 0 to7 do
       (1) eight-lane multiply C[bits 127:0] = A*F
       (eight 16-bit multiplies)
       (2) set D[bits 15:0] = sum of the four 16-bit values
       in C[bits 63:0] (reduction of four 16-bit values)
          (3) set D[bits 31:16] = sum of the four 16-bit
          values in C[bits 127:64] (reduction of four 16-
          bit values)
          (4) store D[bits 31:0] to sig_out (32-bit store)
          (5) set A = A shifted 16 bits to the left
          (6) set E = B shifted 112 shifts to the right
          (7) set A = A OR E
          (8) set B = B shifted 16 bits to the left
 end for
end for
```

# 4

# Solutions

## 4.1

**4.1.1** The values of the signals are as follows:

| RegWrite | MemRead | ALUMux | MemWrite | ALUop | RegMux | Branch |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 (Imm) | 1 | ADD | X | 0 |

ALUMux is the control signal that controls the Mux at the ALU input, 0 (Reg) selects the output of the register file, and 1 (Imm) selects the immediate from the instruction word as the second input to the ALU.

RegMux is the control signal that controls the Mux at the Data input to the register file, 0 (ALU) selects the output of the ALU, and 1 (Mem) selects the output of memory.

A value of X is a "don't care" (does not matter if signal is 0 or 1)

**4.1.2** All except branch Add unit and write port of the Registers

**4.1.3** Outputs that are not used: Branch Add, write port of Registers

No outputs: None (all units produce outputs)

## 4.2

**4.2.1** This instruction uses instruction memory, both register read ports, the ALU to add Rd and Rs together, data memory, and write port in Registers.

**4.2.2** None. This instruction can be implemented using existing blocks.

**4.2.3** None. This instruction can be implemented without adding new control signals. It only requires changes in the Control logic.

## 4.3

**4.3.1** Clock cycle time is determined by the critical path, which for the given latencies happens to be to get the data value for the load instruction: I-Mem (read instruction), Regs (takes longer than Control), Mux (select ALU input), ALU, Data Memory, and Mux (select value from memory to be written into Registers). The latency of this path is 400 ps + 200 ps + 30 ps + 120 ps + 350 ps + 30 ps = 1130 ps. 1430 ps (1130 ps + 300 ps, ALU is on the critical path).

**4.3.2** The speedup comes from changes in clock cycle time and changes to the number of clock cycles we need for the program: We need 5% fewer cycles for a program, but cycle time is 1430 instead of 1130, so we have a speedup of (1/0.95)*(1130/1430) = 0.83, which means we actually have a slowdown.

**4.3.3** The cost is always the total cost of all components (not just those on the critical path, so the original processor has a cost of I-Mem, Regs, Control, ALU, D-Mem, 2 Add units and 3 Mux units, for a total cost of 1000 + 200 + 500 + 100 + 2000 + 2*30 + 3*10 = 3890.

We will compute cost relative to this baseline. The performance relative to this baseline is the speedup we previously computed, and our cost/performance relative to the baseline is as follows:

New Cost: 3890 + 600 = 4490

Relative Cost: 4490/3890 = 1.15

Cost/Performance: 1.15/0.83 = 1.39. We are paying significantly more for significantly worse performance; the cost/performance is a lot worse than with the unmodified processor.

## 4.4

**4.4.1** I-Mem takes longer than the Add unit, so the clock cycle time is equal to the latency of the I-Mem:

200 ps

**4.4.2** The critical path for this instruction is through the instruction memory, Sign-extend and Shift-left-2 to get the offset, Add unit to compute the new PC, and Mux to select that value instead of PC+4. Note that the path through the other Add unit is shorter, because the latency of I-Mem is longer that the latency of the Add unit. We have:

200 ps + 15 ps + 10 ps + 70 ps + 20 ps = 315 ps

**4.4.3** Conditional branches have the same long-latency path that computes the branch address as unconditional branches do. Additionally, they have a long-latency path that goes through Registers, Mux, and ALU to compute the PCSrc condition. The critical path is the longer of the two, and the path through PCSrc is longer for these latencies:

200 ps + 90 ps + 20 ps + 90 ps + 20 ps = 420 ps

**4.4.4** PC-relative branches.

**4.4.5** PC-relative unconditional branch instructions. We saw in part c that this is not on the critical path of conditional branches, and it is only needed for PC-relative branches. Note that MIPS does not have actual unconditional branches (bne zero,zero,Label plays that role so there is no need for unconditional branch opcodes) so for MIPS the answer to this question is actually "None".

**4.4.6** Of the two instructions (BNE and ADD), BNE has a longer critical path so it determines the clock cycle time. Note that every path for ADD is shorter than or equal to the corresponding path for BNE, so changes in unit latency

will not affect this. As a result, we focus on how the unit's latency affects the critical path of BNE.

This unit is not on the critical path, so the only way for this unit to become critical is to increase its latency until the path for address computation through sign extend, shift left, and branch add becomes longer than the path for PCSrc through registers, Mux, and ALU. The latency of Regs, Mux, and ALU is 200 ps and the latency of Sign-extend, Shift-left-2, and Add is 95 ps, so the latency of Shift-left-2 must be increased by 105 ps or more for it to affect clock cycle time.

## 4.5

**4.5.1** The data memory is used by LW and SW instructions, so the answer is:

25% + 10% = 35%

**4.5.2** The sign-extend circuit is actually computing a result in every cycle, but its output is ignored for ADD and NOT instructions. The input of the sign-extend circuit is needed for ADDI (to provide the immediate ALU operand), BEQ (to provide the PC-relative offset), and LW and SW (to provide the offset used in addressing memory) so the answer is:

20% + 25% + 25% + 10% = 80%

## 4.6

**4.6.1** To test for a stuck-at-0 fault on a wire, we need an instruction that puts that wire to a value of 1 and has a different result if the value on the wire is stuck at zero:

If this signal is stuck at zero, an instruction that writes to an odd-numbered register will end up writing to the even-numbered register. So if we place a value of zero in R30 and a value of 1 in R31, and then execute ADD R31,R30,R30 the value of R31 is supposed to be zero. If bit 0 of the Write Register input to the Registers unit is stuck at zero, the value is written to R30 instead and R31 will be 1.

**4.6.2** The test for stuck-at-zero requires an instruction that sets the signal to 1, and the test for stuck-at-1 requires an instruction that sets the signal to 0. Because the signal cannot be both 0 and 1 in the same cycle, we cannot test the same signal simultaneously for stuck-at-0 and stuck-at-1 using only one instruction. The test for stuck-at-1 is analogous to the stuck-at-0 test:

We can place a value of zero in R31 and a value of 1 in R30, then use ADD R30,R31,R31 which is supposed to place 0 in R30. If this signal is stuck-at-1, the write goes to R31 instead, so the value in R30 remains 1.

**4.6.3** We need to rewrite the program to use only odd-numbered registers.

**4.6.4** To test for this fault, we need an instruction whose MemRead is 1, so it has to be a load. The instruction also needs to have RegDst set to 0, which is the case for loads. Finally, the instruction needs to have a different result if

MemRead is set to 0. For a load, MemRead=0 result in not reading memory, so the value placed in the register is "random" (whatever happened to be at the output of the memory unit). Unfortunately, this "random" value can be the same as the one already in the register, so this test is not conclusive.

**4.6.5** To test for this fault, we need an instruction whose Jump is 1, so it has to be the jump instruction. However, for the jump instruction the RegDst signal is "don't care" because it does not write to any registers, so the implementation may or may not allow us to set RegDst to 0 so we can test for this fault. As a result, we cannot reliably test for this fault.

## 4.7

### 4.7.1

| Sign-extend | Jump's shift-left-2 |
|---|---|
| 00000000000000000000000000010100 | 00011000100000000000001010000 |

### 4.7.2

| ALUOp[1-0] | Instruction[5-0] |
|---|---|
| 00 | 010100 |

### 4.7.3

| New PC | Path |
|---|---|
| PC+4 | PC to Add (PC+4) to branch Mux to jump Mux to PC |

### 4.7.4

| WrReg Mux | ALU Mux | Mem/ALU Mux | Branch Mux | Jump Mux |
|---|---|---|---|---|
| 2 or 0 (RegDst is X) | 20 | X | PC+4 | PC+4 |

### 4.7.5

| ALU | Add (PC+4) | Add (Branch) |
|---|---|---|
| -3 and 20 | PC and 4 | PC+4 and 20*4 |

### 4.7.6

| Read Register 1 | Read Register 2 | Write Register | Write Data | RegWrite |
|---|---|---|---|---|

## 4.8

### 4.8.1

| Pipelined | Single-cycle |
|---|---|
| 350 ps | 1250 ps |

### 4.8.2

| Pipelined | Single-cycle |
|---|---|
| 1750 ps | 1250 ps |

### 4.8.3

| Stage to split | New clock cycle time |
|---|---|
| ID | 300 ps |

### 4.8.4

| | |
|---|---|
| a. | 35% |

### 4.8.5

| | |
|---|---|
| a. | 65% |

**4.8.6** We already computed clock cycle times for pipelined and single cycle organizations, and the multi-cycle organization has the same clock cycle time as the pipelined organization. We will compute execution times relative to the pipelined organization. In single-cycle, every instruction takes one (long) clock cycle. In pipelined, a long-running program with no pipeline stalls completes one instruction in every cycle. Finally, a multi-cycle organization completes a LW in 5 cycles, a SW in 4 cycles (no WB), an ALU instruction in 4 cycles (no MEM), and a BEQ in 4 cycles (no WB). So we have the speedup of pipeline

| | Multi-cycle execution time is X times pipelined execution time, where X is: | Single-cycle execution time is X times pipelined execution time, where X is: |
|---|---|---|
| a. | 0.20*5+0.80*4=4.20 | 1250 ps/350 ps=3.57 |

### 4.9

### 4.9.1

| Instruction sequence | Dependences |
|---|---|
| I1: OR R1,R2,R3 | RAW on R1 from I1 to I2 and I3 |
| I2: OR R2,R1,R4 | RAW on R2 from I2 to I3 |
| I3: OR R1,R1,R2 | WAR on R2 from I1 to I2 |
| | WAR on R1 from I2 to I3 |
| | WAW on R1 from I1 to I3 |

**4.9.2** In the basic five-stage pipeline WAR and WAW dependences do not cause any hazards. Without forwarding, any RAW dependence between an instruction and the next two instructions (if register read happens in the second half of the clock cycle and the register write happens in the first half). The code that eliminates these hazards by inserting NOP instructions is:

| Instruction sequence | |
|---|---|
| OR R1,R2,R3 | |
| NOP | Delay I2 to avoid RAW hazard on R1 from I1 |
| NOP | |
| OR R2,R1,R4 | |
| NOP | Delay I3 to avoid RAW hazard on R2 from I2 |
| NOP | |
| OR R1,R1,R2 | |

**4.9.3** With full forwarding, an ALU instruction can forward a value to EX stage of the next instruction without a hazard. However, a load cannot forward to the EX stage of the next instruction (by can to the instruction after that). The code that eliminates these hazards by inserting NOP instructions is:

| Instruction sequence | |
|---|---|
| OR R1,R2,R3 | |
| OR R2,R1,R4 | No RAW hazard on R1 from I1 (forwarded) |
| OR R1,R1,R2 | No RAW hazard on R2 from I2 (forwarded) |

**4.9.4** The total execution time is the clock cycle time times the number of cycles. Without any stalls, a three-instruction sequence executes in 7 cycles (5 to complete the first instruction, then one per instruction). The execution without forwarding must add a stall for every NOP we had in 4.9.2, and execution forwarding must add a stall cycle for every NOP we had in 4.9.3. Overall, we get:

| No forwarding | With forwarding | Speedup due to forwarding |
|---|---|---|
| (7 + 4)*180 ps = 1980 ps | 7*240 ps = 1680 ps | 1.18 |

**4.9.5** With ALU-ALU-only forwarding, an ALU instruction can forward to the next instruction, but not to the second-next instruction (because that would be forwarding from MEM to EX). A load cannot forward at all, because it determines the data value in MEM stage, when it is too late for ALU-ALU forwarding. We have:

| Instruction sequence | |
|---|---|
| OR R1,R2,R3 | |
| OR R2,R1,R4 | ALU-ALU forwarding of R1 from I1 |
| OR R1,R1,R2 | ALU-ALU forwarding of R2 from I2 |

**4.9.6**

| No forwarding | With ALU-ALU forwarding only | Speedup with ALU-ALU forwarding |
|---|---|---|
| (7 + 4)*180 ps = 1980 ps | 7*210 ps = 1470 ps | 1.35 |

## 4.10

**4.10.1** In the pipelined execution shown below, *** represents a stall when an instruction cannot be fetched because a load or store instruction is using the memory in that cycle. Cycles are represented from left to right, and for each instruction we show the pipeline stage it is in during that cycle:

| Instruction | Pipeline Stage | Cycles |
|---|---|---|
| SW R16,12(R6)<br>LW R16,8(R6)<br>BEQ R5,R4,Lbl<br>ADD R5,R1,R4<br>SLT R5,R15,R4 | IF ID EX MEM WB<br>IF ED EX MEM WB<br>IF ID EX MEM WB<br>*** *** IF ID EX MEM WB<br>IF ID EX MEM WB | 11 |

We can not add NOPs to the code to eliminate this hazard – NOPs need to be fetched just like any other instructions, so this hazard must be addressed with a hardware hazard detection unit in the processor.

**4.10.2** This change only saves one cycle in an entire execution without data hazards (such as the one given). This cycle is saved because the last instruction finishes one cycle earlier (one less stage to go through). If there were data hazards from loads to other instructions, the change would help eliminate some stall cycles.

| Instructions Executed | Cycles with 5 stages | Cycles with 4 stages | Speedup |
|---|---|---|---|
| 5 | 4 + 5 = 9 | 3 + 5 = 8 | 9/8 = 1.13 |

**4.10.3** Stall-on-branch delays the fetch of the next instruction until the branch is executed. When branches execute in the EXE stage, each branch causes two stall cycles. When branches execute in the ID stage, each branch only causes one stall cycle. Without branch stalls (e.g., with perfect branch prediction) there are no stalls, and the execution time is 4 plus the number of executed instructions. We have:

| Instructions Executed | Branches Executed | Cycles with branch in EXE | Cycles with branch in ID | Speedup |
|---|---|---|---|---|
| 5 | 1 | 4 + 5 + 1*2 = 11 | 4 + 5 + 1*1 = 10 | 11/10 = 1.10 |

**4.10.4** The number of cycles for the (normal) 5-stage and the (combined EX/MEM) 4-stage pipeline is already computed in 4.10.2. The clock cycle time is equal to the latency of the longest-latency stage. Combining EX and MEM stages affects clock time only if the combined EX/MEM stage becomes the longest-latency stage:

| Cycle time with 5 stages | Cycle time with 4 stages | Speedup |
|---|---|---|
| 200 ps (IF) | 210 ps (MEM + 20 ps) | (9*200)/(8*210) = 1.07 |

**4.10.5**

| New ID latency | New EX latency | New cycle time | Old cycle time | Speedup |
|---|---|---|---|---|
| 180 ps | 140 ps | 200 ps (IF) | 200 ps (IF) | (11*200)/(10*200) = 1.10 |

**4.10.6** The cycle time remains unchanged: a 20 ps reduction in EX latency has no effect on clock cycle time because EX is not the longest-latency stage. The change does affect execution time because it adds one additional stall cycle to each branch. Because the clock cycle time does not improve but

the number of cycles increases, the speedup from this change will be below 1 (a slowdown). In 4.10.3 we already computed the number of cycles when branch is in EX stage. We have:

| | Cycles with branch in EX | Execution time (branch in EX) | Cycles with branch in MEM | Execution time (branch in MEM) | Speedup |
|---|---|---|---|---|---|
| a. | 4 + 5 + 1*2 = 11 | 11*200 ps = 2200 ps | 4 + 5 + 1*3 = 12 | 12*200 ps = 2400 ps | 0.92 |

## 4.11

### 4.11.1

```
LW  R1,0(R1)     | WB
LW  R1,0(R1)     | EX  MEM WB
BEQ R1,R0,Loop   | ID  *** EX  MEM WB
LW  R1,0(R1)     | IF  *** ID  EX  MEM WB
AND R1,R1,R2     |         IF  ID  *** EX  MEM WB
LW  R1,0(R1)     |             IF  *** ID  EX  MEM
LW  R1,0(R1)     |                     IF  ID  ***
BEQ R1,R0,Loop   |                         IF  ***
```

**4.11.2** In a particular clock cycle, a pipeline stage is not doing useful work if it is stalled or if the instruction going through that stage is not doing any useful work there. In the pipeline execution diagram from 4.11.1, a stage is stalled if its name is not shown for a particular cycles, and stages in which the particular instruction is not doing useful work are marked in blue. Note that a BEQ instruction is doing useful work in the MEM stage, because it is determining the correct value of the next instruction's PC in that stage. We have:

| Cycles per loop iteration | Cycles in which all stages do useful work | % of cycles in which all stages do useful work |
|---|---|---|
| 8 | 0 | 0% |

## 4.12

**4.12.1** Dependences to the 1st next instruction result in 2 stall cycles, and the stall is also 2 cycles if the dependence is to both 1st and 2nd next instruction. Dependences to only the 2nd next instruction result in one stall cycle. We have:

| CPI | Stall Cycles |
|---|---|
| 1 + 0.35*2 + 0.15*1 = 1.85 | 46% (0.85/1.85) |

**4.12.2** With full forwarding, the only RAW data dependences that cause stalls are those from the MEM stage of one instruction to the 1st next instruction. Even this dependences causes only one stall cycle, so we have:

| CPI | Stall Cycles |
|---|---|
| 1 + 0.20 = 1.20 | 17% (0.20/1.20) |

**4.12.3** With forwarding only from the EX/MEM register, EX to 1st dependences can be satisfied without stalls but any other dependences (even when together with EX to 1st) incur a one-cycle stall. With forwarding only from the MEM/WB register, EX to 2nd dependences incur no stalls. MEM to 1st dependences still incur a one-cycle stall, and EX to 1st dependences now incur one stall cycle because we must wait for the instruction to complete the MEM stage to be able to forward to the next instruction. We compute stall cycles per instructions for each case as follows:

| EX/MEM | MEM/WB | Fewer stall cycles with |
|---|---|---|
| 0.2 + 0.05 + 0.1 + 0.1 = 0.45 | 0.05 + 0.2 + 0.1 = 0.35 | MEM/WB |

**4.12.4** In 4.12.1 and 4.12.2 we have already computed the CPI without forwarding and with full forwarding. Now we compute time per instruction by taking into account the clock cycle time:

| Without forwarding | With forwarding | Speedup |
|---|---|---|
| 1.85*150 ps = 277.5 ps | 1.20*150 ps = 180 ps | 1.54 |

**4.12.5** We already computed the time per instruction for full forwarding in 4.12.4. Now we compute time-per instruction with time-travel forwarding and the speedup over full forwarding:

| With full forwarding | Time-travel forwarding | Speedup |
|---|---|---|
| 1.20*150 ps = 180 ps | 1*250 ps = 250 ps | 0.72 |

**4.12.6**

| EX/MEM | MEM/WB | Shorter time per instruction with |
|---|---|---|
| 1.45*150 ps = 217.5 | 1.35*150 ps = 202.5 ps | MEM/WB |

## 4.13

### 4.13.1

```
ADD R5,R2,R1
NOP
NOP
LW R3,4(R5)
LW R2,0(R2)
NOP
OR R3,R5,R3
NOP
NOP
SW R3,0(R5)
```

**4.13.2** We can move up an instruction by swapping its place with another instruction that has no dependences with it, so we can try to fill some NOP slots with such instructions. We can also use R7 to eliminate WAW or WAR dependences so we can have more instructions to move up.

| | |
|---|---|
| I1: ADD R5,R2,R1<br>I3: LW R2,0(R2)<br>NOP<br>I2: LW R3,4(R5)<br>NOP<br>NOP<br>I4: OR R3,R5,R3<br>NOP<br>NOP<br>I5: SW R3,0(R5) | Moved up to fill NOP slot<br><br><br>Had to add another NOP here, so there is no performance gain |

**4.13.3** With forwarding, the hazard detection unit is still needed because it must insert a one-cycle stall whenever the load supplies a value to the instruction that immediately follows that load. Without the hazard detection unit, the instruction that depends on the immediately preceding load gets the stale value the register had before the load instruction.

Code executes correctly (for both loads, there is no RAW dependence between the load and the next instruction).

**4.13.4** The outputs of the hazard detection unit are PCWrite, IF/IDWrite, and ID/EXZero (which controls the Mux after the output of the Control unit). Note that IF/IDWrite is always equal to PCWrite, and ED/ExZero is always the opposite of PCWrite. As a result, we will only show the value of PCWrite for each cycle. The outputs of the forwarding unit is ALUin1 and ALUin2, which control Muxes that select the first and second input of the ALU. The three possible values for ALUin1 or ALUin2 are 0 (no forwarding), 1 (forward ALU output from previous instruction), or 2 (forward data value for second-previous instruction). We have:

| Instruction sequence | First five cycles<br>1 2 3 4 5 | Signals |
|---|---|---|
| ADD R5,R2,R1<br>LW R3,4(R5)<br>LW R2,0(R2)<br>OR R3,R5,R3<br>SW R3,0(R5) | IF ID EX MEM WB<br>IF ID EX MEM<br>IF ID EX<br>IF ID<br>IF | 1: PCWrite=1, ALUin1=X, ALUin2=X<br>2: PCWrite=1, ALUin1=X, ALUin2=X<br>3: PCWrite=1, ALUin1=0, ALUin2=0<br>4: PCWrite=1, ALUin1=1, ALUin2=0<br>5: PCWrite=1, ALUin1=0, ALUin2=0 |

**4.13.5** The instruction that is currently in the ID stage needs to be stalled if it depends on a value produced by the instruction in the EX or the instruction in the MEM stage. So we need to check the destination register of these two instructions. For the instruction in the EX stage, we need to check Rd for R-type instructions and Rd for loads. For the instruction in the MEM stage, the destination register is already selected (by the Mux in the EX stage) so we need to check that register number (this is the bottommost output of the EX/MEM pipeline register). The additional inputs to the hazard detection unit

are register Rd from the ID/EX pipeline register and the output number of the output register from the EX/MEM pipeline register. The Rt field from the ID/EX register is already an input of the hazard detection unit in Figure 4.60.

No additional outputs are needed. We can stall the pipeline using the three output signals that we already have.

**4.13.6** As explained for part e, we only need to specify the value of the PCWrite signal, because IF/IDWrite is equal to PCWrite and the ID/EXzero signal is its opposite.We have:

| Instruction sequence | First five cycles<br>1 2 3 4 5 | Signals |
|---|---|---|
| ADD R5,R2,R1<br>LW R3,4(R5)<br>LW R2,0(R2)<br>OR R3,R5,R3<br>SW R3,0(R5) | IF ID EX MEM WB<br>IF ID *** ***<br>IF *** ***<br>***<br> | 1: PCWrite=1<br>2: PCWrite=1<br>3: PCWrite=1<br>4: PCWrite=0<br>5: PCWrite=0 |

## 4.14

### 4.14.1

| Executed Instructions | Pipeline Cycles<br>1 2 3 4 5 6 7 8 9 10 11 12 13 14 |
|---|---|
| LW R2,0(R1)<br>BEQ R2,R0,Label2 (NT)<br>LW R3,0(R2)<br>BEQ R3,R0,Label1 (T)<br>BEQ R2,R0,Label2 (T)<br>SW R1,0(R2) | IF ID EX MEM WB<br>IF ID *** EX MEM WB<br>IF ID EX MEM WB<br>IF ID *** EX MEM WB<br>IF *** ID EX MEM WB<br>IF ID EX MEM WB |

### 4.14.2

| Executed Instructions | Pipeline Cycles<br>1 2 3 4 5 6 7 8 9 10 11 12 13 14 |
|---|---|
| LW R2,0(R1)<br>BEQ R2,R0,Label2 (NT)<br>LW R3,0(R2)<br>BEQ R3,R0,Label1 (T)<br>ADD R1,R3,R1<br>BEQ R2,R0,Label2 (T)<br>LW R3,0(R2)<br>SW R1,0(R2) | IF ID EX MEM WB<br>IF ID *** EX MEM WB<br>IF *** ID EX MEM WB<br>IF ID EX MEM WB<br>IF ID EX MEM WB<br>IF ID EX MEM WB<br>IF ID EX MEM WB<br>IF ID EX MEM WB |

### 4.14.3

```
        LW R2,0(R1)
Label1: BEZ R2,Label2 ; Not taken once, then taken
        LW R3,0(R2)
        BEZ R3,Label1 ; Taken
        ADD R1,R3,R1
Label2: SW R1,0(R2)
```

**4.14.4** The hazard detection logic must detect situations when the branch depends on the result of the previous R-type instruction, or on the result of two previous loads. When the branch uses the values of its register operands in its ID stage, the R-type instruction's result is still being generated in the EX stage. Thus we must stall the processor and repeat the ID stage of the branch in the next cycle. Similarly, if the branch depends on a load that immediately precedes it, the result of the load is only generated two cycles after the branch enters the ID stage, so we must stall the branch for two cycles. Finally, if the branch depends on a load that is the second-previous instruction, the load is completing its MEM stage when the branch is in its ID stage, so we must stall the branch for one cycle. In all three cases, the hazard is a data hazard.

Note that in all three cases we assume that the values of preceding instructions are forwarded to the ID stage of the branch if possible.

**4.14.5** For part a we have already shows the pipeline execution diagram for the case when branches are executed in the EX stage. The following is the pipeline diagram when branches are executed in the ID stage, including new stalls due to data dependences described for part d:

| Executed Instructions | Pipeline Cycles 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 |
|---|---|
| LW R2,0(R1) | IF ID EX MEM WB |
| BEQ R2,R0,Label2 (NT) | IF *** ***ID EX MEM WB |
| LW R3,0(R2) | IF ID EX MEM WB |
| BEQ R3,R0,Label1 (T) | IF *** *** ID EX MEM WB |
| BEQ R2,R0,Label2 (T) | IF ID EX MEM WB |
| SW R1,0(R2) | IF ID EX MEM WB |

Now the speedup can be computed as:

$14/15 = 0.93$

**4.14.6** Branch instructions are now executed in the ID stage. If the branch instruction is using a register value produced by the immediately preceding instruction, as we described for part d the branch must be stalled because the preceding instruction is in the EX stage when the branch is already using the stale register values in the ID stage. If the branch in the ID stage depends on an R-type instruction that is in the MEM stage, we need forwarding to ensure correct execution of the branch. Similarly, if the branch in the ID stage depends on an R-type of load instruction in the WB stage, we need forwarding to ensure correct execution of the branch. Overall, we need another forwarding unit that takes the same inputs as the one that forwards to the EX stage. The new forwarding unit should control two Muxes placed right before the branch comparator. Each Mux selects between the value read from Registers, the ALU output from the EX/MEM pipeline register, and the data value from the MEM/WB pipeline register. The complexity of the new forwarding unit is the same as the complexity of the existing one.

## 4.15

**4.15.1** Each branch that is not correctly predicted by the always-taken predictor will cause 3 stall cycles, so we have:

| Extra CPI |
|---|
| 3*(1 – 0.45)*0.25 = 0.41 |

**4.15.2** Each branch that is not correctly predicted by the always-not-taken predictor will cause 3 stall cycles, so we have:

| Extra CPI |
|---|
| 3*(1 – 0.55)*0.25 = 0.34 |

**4.15.3** Each branch that is not correctly predicted by the 2-bit predictor will cause 3 stall cycles, so we have:

| Extra CPI |
|---|
| 3*(1 – 0.85)*0.25 = 0.113 |

**4.15.4** Correctly predicted branches had CPI of 1 and now they become ALU instructions whose CPI is also 1. Incorrectly predicted instructions that are converted also become ALU instructions with a CPI of 1, so we have:

| CPI without conversion | CPI with conversion | Speedup from conversion |
|---|---|---|
| 1 + 3*(1-0.85)*0.25 = 1.113 | 1 + 3*(1-0.85)*0.25*0.5 = 1.056 | 1.113/1.056 = 1.054 |

**4.15.5** Every converted branch instruction now takes an extra cycle to execute, so we have:

| CPI without conversion | Cycles per original instruction with conversion | Speedup from conversion |
|---|---|---|
| 1.113 | 1 + (1 + 3*(1 – 0.85))*0.25*0.5 = 1.181 | 1.113/1.181 = 0.94 |

**4.15.6** Let the total number of branch instructions executed in the program be B. Then we have:

| Correctly predicted | Correctly predicted non-loop-back | Accuracy on non-loop-back branches |
|---|---|---|
| B*0.85 | B*0.05 | (B*0.05)/(B*0.20) = 0.25 (25%) |

## 4.16

### 4.16.1

| Always Taken | Always not-taken |
|---|---|
| 3/5 = 60% | 2/5 = 40% |

### 4.16.2

| Outcomes | Predictor value at time of prediction | Correct or Incorrect | Accuracy |
|---|---|---|---|
| T, NT, T, T | 0,1,0,1 | I,C,I,I | 25% |

**4.16.3** The first few recurrences of this pattern do not have the same accuracy as the later ones because the predictor is still warming up. To determine the accuracy in the "steady state", we must work through the branch predictions until the predictor values start repeating (i.e., until the predictor has the same value at the start of the current and the next recurrence of the pattern).

| Outcomes | Predictor value at time of prediction | Correct or Incorrect (in steady state) | Accuracy in steady state |
|---|---|---|---|
| T, NT, T, T, NT | 1st occurrence: 0,1,0,1,2<br>2nd occurrence: 1,2,1,2,3<br>3rd occurrence: 2,3,2,3,3<br>4th occurrence: 2,3,2,3,3 | C,I,C,C,I | 60% |

**4.16.4** The predictor should be an N-bit shift register, where N is the number of branch outcomes in the target pattern. The shift register should be initialized with the pattern itself (0 for NT, 1 for T), and the prediction is always the value in the leftmost bit of the shift register. The register should be shifted after each predicted branch.

**4.16.5** Since the predictor's output is always the opposite of the actual outcome of the branch instruction, the accuracy is zero.

**4.16.6** The predictor is the same as in part d, except that it should compare its prediction to the actual outcome and invert (logical NOT) all the bits in the shift register if the prediction is incorrect. This predictor still always perfectly predicts the given pattern. For the opposite pattern, the first prediction will be incorrect, so the predictor's state is inverted and after that the predictions are always correct. Overall, there is no warm-up period for the given pattern, and the warm-up period for the opposite pattern is only one branch.

### 4.17

### 4.17.1

| Instruction 1 | Instruction 2 |
|---|---|
| Invalid target address (EX) | Invalid data address (MEM) |

**4.17.2** The Mux that selects the next PC must have inputs added to it. Each input is a constant address of an exception handler. The exception detectors

must be added to the appropriate pipeline stage and the outputs of these detectors must be used to control the pre-PC Mux, and also to convert to NOPs instructions that are already in the pipeline behind the exception-triggering instruction.

**4.17.3** Instructions are fetched normally until the exception is detected. When the exception is detected, all instructions that are in the pipeline after the first instruction must be converted to NOPs. As a result, the second instruction never completes and does not affect pipeline state. In the cycle that immediately follows the cycle in which the exception is detected, the processor will fetch the first instruction of the exception handler.

**4.17.4** This approach requires us to fetch the address of the handler from memory. We must add the code of the exception to the address of the exception vector table, read the handler's address from memory, and jump to that address. One way of doing this is to handle it like a special instruction that computer the address in EX, loads the handler's address in MEM, and sets the PC in WB.

**4.17.5** We need a special instruction that allows us to move a value from the (exception) Cause register to a general-purpose register. We must first save the general-purpose register (so we can restore it later), load the Cause register into it, add the address of the vector table to it, use the result as an address for a load that gets the address of the right exception handler from memory, and finally jump to that handler.

## 4.18

### 4.18.1

```
       ADD R5,R0,R0
Again: BEQ R5,R6,End
       ADD R10,R5,R1
       LW R11,0(R10)
       LW R10,1(R10)
       SUB R10,R11,R10
       ADD R11,R5,R2
       SW R10,0(R11)
       ADDI R5,R5,2
       BEW R0,R0,Again
End:
```

### 4.18.2

| Instructions | Pipeline |
|---|---|
| ADD R5,R0,R0 | IF ID EX ME WB |
| BEQ R5,R6,End | IF ID ** EX ME WB |
| ADD R10,R5,R1 | IF ** ID EX ME WB |
| LW R11,0(R10) | IF ** ID ** EX ME WB |
| LW R10,1(R10) | IF ** ID EX ME WB |
| SUB R10,R11,R10 | IF ** ID ** ** EX ME WB |
| ADD R11,R5,R2 | IF ** ** ID EX ME WB |
| SW R10,0(R11) | IF ** ** ID ** EX ME WB |
| ADDI R5,R5,2 | IF ** ID EX ME WB |
| BEW R0,R0,Again | IF ** ID ** EX ME WB |
| BEQ R5,R6,End | IF ** ID EX ME WB |
| ADD R10,R5,R1 | IF ** ID ** EX ME WB |
| LW R11,0(R10) | IF ** ID EX ME WB |
| LW R10,1(R10) | IF ** ID ** EX ME WB |
| SUB R10,R11,R10 | IF ** ID ** EX ME WB |
| ADD R11,R5,R2 | IF ** ID ** ** EX ME WB |
| SW R10,0(R11) | IF ** ** ID EX ME WB |
| ADDI R5,R5,2 | IF ** ** ID EX ME WB |
| BEW R0,R0,Again | IF ID EX ME WB |
| BEQ R5,R6,End | IF ID ** EX ME WB |

**4.18.3** The only way to execute 2 instructions fully in parallel is for a load/store to execute together with another instruction. To achieve this, around each load/store instruction we will try to put non-load/store instructions that have no dependences with the load/store.

```
       ADD R5,R0,R0
Again: ADD R10,R5,R1
       BEQ R5,R6,End
       LW R11,0(R10)
       ADD R12,R5,R2
       LW R10,1(R10)
       ADDI R5,R5,2
       SUB R10,R11,R10
       SW R10,0(R12)
       BEQ R0,R0,Again
End:
```

Note that we are now computing a+i before we check whether we should continue the loop. This is OK because we are allowed to "trash" R10. If we exit the loop one extra instruction is executed, but if we stay in the loop we allow both of the memory instructions to execute in parallel with other instructions

### 4.18.4

| Instructions | Pipeline |
|---|---|
| `ADD R5,R0,R0` | `IF ID EX ME WB` |
| `ADD R10,R5,R1` | `IF ID ** EX ME WB` |
| `BEQ R5,R6,End` | `IF ** ID EX ME WB` |
| `LW R11,0(R10)` | `IF ** ID EX ME WB` |
| `ADD R12,R5,R2` | `IF ID EX ME WB` |
| `LW R10,1(R10)` | `IF ID EX ME WB` |
| `ADDI R5,R5,2` | `IF ID EX ME WB` |
| `SUB R10,R11,R10` | `IF ID ** EX ME WB` |
| `SW R10,0(R12)` | `IF ** ID EX ME WB` |
| `BEQ R0,R0,Again` | `IF ** ID EX ME WB` |
| `ADD R10,R5,R1` | `IF ** ID EX ME WB` |
| `BEQ R5,R6,End` | `IF ** ID ** EX ME WB` |
| `LW R11,0(R10)` | `IF ** ID EX ME WB` |
| `ADD R12,R5,R2` | `IF ** ID EX ME WB` |
| `LW R10,1(R10)` | `IF ID EX ME WB` |
| `ADDI R5,R5,2` | `IF ID EX ME WB` |
| `SUB R10,R11,R10` | `IF ID ** EX ME WB` |
| `SW R10,0(R12)` | `IF ID ** EX ME WB` |
| `BEQ R0,R0,Again` | `IF ** ID EX ME WB` |
| `ADD R10,R5,R1` | `IF ** ID ** EX ME WB` |
| `BEQ R5,R6,End` | `IF ** ID EX ME WB` |

### 4.18.5

| CPI for 1-issue | CPI for 2-issue | Speedup |
|---|---|---|
| 1.11 (10 cycles per 9 instructions). There is 1 stall cycle in each iteration due to a data hazard between the second `LW` and the next instruction (`SUB`). | 1.06 (19 cycles per 18 instructions). Neither of the two `LW` instructions can execute in parallel with another instruction, and `SUB` stalls because it depends on the second `LW`. The `SW` instruction executes in parallel with `ADDI` in even-numbered iterations. | 1.05 |

### 4.18.6

| CPI for1-issue | CPI for 2-issue | Speedup |
|---|---|---|
| 1.11 | 0.83 (15 cycles per 18 instructions). In all iterations, `SUB` is stalled because it depends on the second `LW`. The only instructions that execute in odd-numbered iterations as a pair are `ADDI` and `BEQ`. In even-numbered iterations, only the two `LW` instruction cannot execute as a pair. | 1.34 |

### 4.19

**4.19.1** The energy for the two designs is the same: I-Mem is read, two registers are read, and a register is written. We have:

140 pJ + 2*70 ps + 60 pJ = 340 pJ

**4.19.2** The instruction memory is read for all instructions. Every instruction also results in two register reads (even if only one of those values is actually used). A load instruction results in a memory read and a register write, a store instruction results in a memory write, and all other instructions result in either no register write (e.g., `BEQ`) or a register write. Because the sum of memory read and register write energy is larger than memory write energy, the worst-case instruction is a load instruction. For the energy spent by a load, we have:

| 140 pJ + 2*70 pJ + 60 pJ + 140 pJ = 480 pJ |
|---|

**4.19.3** Instruction memory must be read for every instruction. However, we can avoid reading registers whose values are not going to be used. To do this, we must add RegRead1 and RegRead2 control inputs to the Registers unit to enable or disable each register read. We must generate these control signals quickly to avoid lengthening the clock cycle time. With these new control signals, a LW instruction results in only one register read (we still must read the register used to generate the address), so we have:

| Energy before change | Energy saved by change | % Savings |
|---|---|---|
| 140 pJ + 2*70 pJ + 60 pJ + 140 pJ = 480 pJ | 70 pJ | 14.6% |

**4.19.4** Before the change, the Control unit decodes the instruction while register reads are happening. After the change, the latencies of Control and Register Read cannot be overlapped. This increases the latency of the ID stage and could affect the processor's clock cycle time if the ID stage becomes the longest-latency stage. We have:

| Clock cycle time before change | Clock cycle time after change |
|---|---|
| 250 ps (D-Mem in MEM stage) | No change (150 ps + 90 ps < 250 ps) |

**4.19.5** If memory is read in every cycle, the value is either needed (for a load instruction), or it does not get past the WB Mux (or a non-load instruction that writes to a register), or it does not get written to any register (all other instructions, including stalls). This change does not affect clock cycle time because the clock cycle time must already allow enough time for memory to be read in the MEM stage. It does affect energy: a memory read occurs in every cycle instead of only in cycles when a load instruction is in the MEM stage.

| I-Mem active energy | I-Mem latency | Clock cycle time | Total I-Mem energy | Idle energy % |
|---|---|---|---|---|
| 140 pJ | 200 ps | 250 ps | 140 pJ + 50 ps*0.1*140 pJ/200 ps = 143.5 pJ | 3.5 pJ/143.5 pJ = 2.44% |

# 5

# Solutions

## 5.1

**5.1.1** 4

**5.1.2** I, J

**5.1.3** A[I][J]

**5.1.4** 3596 = 8 × 800/4 × 2−8×8/4 + 8000/4

**5.1.5** I, J

**5.1.6** A(J, I)

## 5.2

### 5.2.1

| Word Address | Binary Address | Tag | Index | Hit/Miss |
|---|---|---|---|---|
| 3 | 0000 0011 | 0 | 3 | M |
| 180 | 1011 0100 | 11 | 4 | M |
| 43 | 0010 1011 | 2 | 11 | M |
| 2 | 0000 0010 | 0 | 2 | M |
| 191 | 1011 1111 | 11 | 15 | M |
| 88 | 0101 1000 | 5 | 8 | M |
| 190 | 1011 1110 | 11 | 14 | M |
| 14 | 0000 1110 | 0 | 14 | M |
| 181 | 1011 0101 | 11 | 5 | M |
| 44 | 0010 1100 | 2 | 12 | M |
| 186 | 1011 1010 | 11 | 10 | M |
| 253 | 1111 1101 | 15 | 13 | M |

### 5.2.2

| Word Address | Binary Address | Tag | Index | Hit/Miss |
|---|---|---|---|---|
| 3 | 0000 0011 | 0 | 1 | M |
| 180 | 1011 0100 | 11 | 2 | M |
| 43 | 0010 1011 | 2 | 5 | M |
| 2 | 0000 0010 | 0 | 1 | H |
| 191 | 1011 1111 | 11 | 7 | M |
| 88 | 0101 1000 | 5 | 4 | M |
| 190 | 1011 1110 | 11 | 7 | H |
| 14 | 0000 1110 | 0 | 7 | M |
| 181 | 1011 0101 | 11 | 2 | H |
| 44 | 0010 1100 | 2 | 6 | M |
| 186 | 1011 1010 | 11 | 5 | M |
| 253 | 1111 1101 | 15 | 6 | M |

### 5.2.3

| Word Address | Binary Address | Tag | Cache 1 | | Cache 2 | | Cache 3 | |
|---|---|---|---|---|---|---|---|---|
| | | | index | hit/miss | index | hit/miss | index | hit/miss |
| 3 | 0000 0011 | 0 | 3 | M | 1 | M | 0 | M |
| 180 | 1011 0100 | 22 | 4 | M | 2 | M | 1 | M |
| 43 | 0010 1011 | 5 | 3 | M | 1 | M | 0 | M |
| 2 | 0000 0010 | 0 | 2 | M | 1 | M | 0 | M |
| 191 | 1011 1111 | 23 | 7 | M | 3 | M | 1 | M |
| 88 | 0101 1000 | 11 | 0 | M | 0 | M | 0 | M |
| 190 | 1011 1110 | 23 | 6 | M | 3 | H | 1 | H |
| 14 | 0000 1110 | 1 | 6 | M | 3 | M | 1 | M |
| 181 | 1011 0101 | 22 | 5 | M | 2 | H | 1 | M |
| 44 | 0010 1100 | 5 | 4 | M | 2 | M | 1 | M |
| 186 | 1011 1010 | 23 | 2 | M | 1 | M | 0 | M |
| 253 | 1111 1101 | 31 | 5 | M | 2 | M | 1 | M |

Cache 1 miss rate = 100%

Cache 1 total cycles = $12 \times 25 + 12 \times 2 = 324$

Cache 2 miss rate = 10/12 = 83%

Cache 2 total cycles = $10 \times 25 + 12 \times 3 = 286$

Cache 3 miss rate = 11/12 = 92%

Cache 3 total cycles = $11 \times 25 + 12 \times 5 = 335$

Cache 2 provides the best performance.

**5.2.4** First we must compute the number of cache blocks in the initial cache configuration. For this, we divide 32 KiB by 4 (for the number of bytes per word) and again by 2 (for the number of words per block). This gives us 4096 blocks and a resulting index field width of 12 bits. We also have a word offset size of 1 bit and a byte offset size of 2 bits. This gives us a tag field size of 32 − 15 = 17 bits. These tag bits, along with one valid bit per block, will require $18 \times 4096 = 73728$ bits or 9216 bytes. The total cache size is thus 9216 + 32768 = 41984 bytes.

The total cache size can be generalized to

totalsize = datasize + (validbitsize + tagsize) × blocks

totalsize = 41984

datasize = blocks × blocksize × wordsize

wordsize = 4

tagsize = 32 − log2(blocks) − log2(blocksize) − log2(wordsize)

validbitsize = 1

Increasing from 2-word blocks to 16-word blocks will reduce the tag size from 17 bits to 14 bits.

In order to determine the number of blocks, we solve the inequality:

41984 <= 64 × blocks + 15 × blocks

Solving this inequality gives us 531 blocks, and rounding to the next power of two gives us a 1024-block cache.

The larger block size may require an increased hit time and an increased miss penalty than the original cache. The fewer number of blocks may cause a higher conflict miss rate than the original cache.

**5.2.5** Associative caches are designed to reduce the rate of conflict misses. As such, a sequence of read requests with the same 12-bit index field but a different tag field will generate many misses. For the cache described above, the sequence 0, 32768, 0, 32768, 0, 32768, …, would miss on every access, while a 2-way set associate cache with LRU replacement, even one with a significantly smaller overall capacity, would hit on every access after the first two.

**5.2.6** Yes, it is possible to use this function to index the cache. However, information about the five bits is lost because the bits are XOR'd, so you must include more tag bits to identify the address in the cache.

## 5.3

**5.3.1** 8

**5.3.2** 32

**5.3.3** 1+ (22/8/32) = 1.086

**5.3.4** 3

**5.3.5** 0.25

**5.3.6** <Index, tag, data>

<$000001_2$, $0001_2$, mem[1024]>

<$000001_2$, $0011_2$, mem[16]>

<$001011_2$, $0000_2$, mem[176]>

<$001000_2$, $0010_2$, mem[2176]>

<$001110_2$, $0000_2$, mem[224]>

<$001010_2$, $0000_2$, mem[160]>

## 5.4

**5.4.1** The L1 cache has a low write miss penalty while the L2 cache has a high write miss penalty. A write buffer between the L1 and L2 cache would hide the write miss latency of the L2 cache. The L2 cache would benefit from write buffers when replacing a dirty block, since the new block would be read in before the dirty block is physically written to memory.

**5.4.2** On an L1 write miss, the word is written directly to L2 without bringing its block into the L1 cache. If this results in an L2 miss, its block must be brought into the L2 cache, possibly replacing a dirty block which must first be written to memory.

**5.4.3** After an L1 write miss, the block will reside in L2 but not in L1. A subsequent read miss on the same block will require that the block in L2 be written back to memory, transferred to L1, and invalidated in L2.

**5.4.4** One in four instructions is a data read, one in ten instructions is a data write. For a CPI of 2, there are 0.5 instruction accesses per cycle, 12.5% of cycles will require a data read, and 5% of cycles will require a data write.

The instruction bandwidth is thus $(0.0030 \times 64) \times 0.5 = 0.096$ bytes/cycle. The data read bandwidth is thus $0.02 \times (0.13+0.050) \times 64 = 0.23$ bytes/cycle. The total read bandwidth requirement is 0.33 bytes/cycle. The data write bandwidth requirement is $0.05 \times 4 = 0.2$ bytes/cycle.

**5.4.5** The instruction and data read bandwidth requirement is the same as in 5.4.4. The data write bandwidth requirement becomes $0.02 \times 0.30 \times (0.13+0.050) \times 64 = 0.069$ bytes/cycle.

**5.4.6** For CPI=1.5 the instruction throughput becomes $1/1.5 = 0.67$ instructions per cycle. The data read frequency becomes $0.25 / 1.5 = 0.17$ and the write frequency becomes $0.10 / 1.5 = 0.067$.

The instruction bandwidth is $(0.0030 \times 64) \times 0.67 = 0.13$ bytes/cycle.

For the write-through cache, the data read bandwidth is $0.02 \times (0.17 +0.067) \times 64 = 0.22$ bytes/cycle. The total read bandwidth is 0.35 bytes/cycle. The data write bandwidth is $0.067 \times 4 = 0.27$ bytes/cycle.

For the write-back cache, the data write bandwidth becomes $0.02 \times 0.30 \times (0.17+0.067) \times 64 = 0.091$ bytes/cycle.

| Address | 0 | 4 | 16 | 132 | 232 | 160 | 1024 | 30 | 140 | 3100 | 180 | 2180 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Line ID | 0 | 0 | 1 | 8 | 14 | 10 | 0 | 1 | 9 | 1 | 11 | 8 |
| Hit/miss | M | H | M | M | M | M | M | H | H | M | M | M |
| Replace | N | N | N | N | N | N | Y | N | N | Y | N | Y |

## 5.5

**5.5.1** Assuming the addresses given as byte addresses, each group of 16 accesses will map to the same 32-byte block so the cache will have a miss rate of 1/16. All misses are compulsory misses. The miss rate is not sensitive to the size of the cache or the size of the working set. It is, however, sensitive to the access pattern and block size.

**5.5.2** The miss rates are 1/8, 1/32, and 1/64, respectively. The workload is exploiting temporal locality.

**5.5.3** In this case the miss rate is 0.

**5.5.4** AMAT for B = 8: $0.040 \times (20 \times 8) = 6.40$

AMAT for B = 16: $0.030 \times (20 \times 16) = 9.60$

AMAT for B = 32: $0.020 \times (20 \times 32) = 12.80$

AMAT for B = 64: $0.015 \times (20 \times 64) = 19.20$

AMAT for B = 128: $0.010 \times (20 \times 128) = 25.60$

B = 8 is optimal.

**5.5.5** AMAT for B = 8: $0.040 \times (24 + 8) = 1.28$

AMAT for B = 16: $0.030 \times (24 + 16) = 1.20$

AMAT for B = 32: $0.020 \times (24 + 32) = 1.12$

AMAT for B = 64: $0.015 \times (24 + 64) = 1.32$

AMAT for B = 128: $0.010 \times (24 + 128) = 1.52$

B = 32 is optimal.

**5.5.6** B=128

## 5.6

### 5.6.1

| | |
|---|---|
| P1 | 1.52 GHz |
| P2 | 1.11 GHz |

### 5.6.2

| | | |
|---|---|---|
| P1 | 6.31 ns | 9.56 cycles |
| P2 | 5.11 ns | 5.68 cycles |

### 5.6.3

| | | |
|---|---|---|
| P1 | 12.64 CPI | 8.34 ns per inst |
| P2 | 7.36 CPI | 6.63 ns per inst |

### 5.6.4

| | | |
|---|---|---|
| 6.50 ns | 9.85 cycles | Worse |

**5.6.5** 13.04

**5.6.6** P1 AMAT = 0.66 ns + 0.08 × 70 ns = 6.26 ns

P2 AMAT = 0.90 ns + 0.06 × (5.62 ns + 0.95 × 70 ns) = 5.23 ns

For P1 to match P2's performance:

5.23 = 0.66 ns + MR × 70 ns

MR = 6.5%

### 5.7

**5.7.1** The cache would have 24 / 3 = 8 blocks per way and thus an index field of 3 bits.

| Word Address | Binary Address | Tag | Index | Hit/Miss | Way 0 | Way 1 | Way 2 |
|---|---|---|---|---|---|---|---|
| 3 | 0000 0011 | 0 | 1 | M | T(1)=0 | | |
| 180 | 1011 0100 | 11 | 2 | M | T(1)=0<br>T(2)=11 | | |
| 43 | 0010 1011 | 2 | 5 | M | T(1)=0<br>T(2)=11<br>T(5)=2 | | |
| 2 | 0000 0010 | 0 | 1 | M | T(1)=0<br>T(2)=11<br>T(5)=2 | T(1)=0 | |
| 191 | 1011 1111 | 11 | 7 | M | T(1)=0<br>T(2)=11<br>T(5)=2<br>T(7)=11 | T(1)=0 | |
| 88 | 0101 1000 | 5 | 4 | M | T(1)=0<br>T(2)=11<br>T(5)=2<br>T(7)=11<br>T(4)=5 | T(1)=0 | |
| 190 | 1011 1110 | 11 | 7 | H | T(1)=0<br>T(2)=11<br>T(5)=2<br>T(7)=11<br>T(4)=5 | T(1)=0 | |
| 14 | 0000 1110 | 0 | 7 | M | T(1)=0<br>T(2)=11<br>T(5)=2<br>T(7)=11<br>T(4)=5 | T(1)=0<br>T(7)=0 | |
| 181 | 1011 0101 | 11 | 2 | H | T(1)=0<br>T(2)=11<br>T(5)=2<br>T(7)=11<br>T(4)=5 | T(1)=0<br>T(7)=0 | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 44 | 0010 1100 | 2 | 6 | M | T(1)=0<br>T(2)=11<br>T(5)=2<br>T(7)=11<br>T(4)=5<br>T(6)=2 | T(1)=0<br>T(7)=0 | |
| 186 | 1011 1010 | 11 | 5 | M | T(1)=0<br>T(2)=11<br>T(5)=2<br>T(7)=11<br>T(4)=5<br>T(6)=2 | T(1)=0<br>T(7)=0<br>T(5)=11 | |
| 253 | 1111 1101 | 15 | 6 | M | T(1)=0<br>T(2)=11<br>T(5)=2<br>T(7)=11<br>T(4)=5<br>T(6)=2 | T(1)=0<br>T(7)=0<br>T(5)=11<br>T(6)=15 | |

**5.7.2** Since this cache is fully associative and has one-word blocks, the word address is equivalent to the tag. The only possible way for there to be a hit is a repeated reference to the same word, which doesn't occur for this sequence.

| Tag | Hit/Miss | Contents |
|---|---|---|
| 3 | M | 3 |
| 180 | M | 3, 180 |
| 43 | M | 3, 180, 43 |
| 2 | M | 3, 180, 43, 2 |
| 191 | M | 3, 180, 43, 2, 191 |
| 88 | M | 3, 180, 43, 2, 191, 88 |
| 190 | M | 3, 180, 43, 2, 191, 88, 190 |
| 14 | M | 3, 180, 43, 2, 191, 88, 190, 14 |
| 181 | M | 181, 180, 43, 2, 191, 88, 190, 14 |
| 44 | M | 181, 44, 43, 2, 191, 88, 190, 14 |
| 186 | M | 181, 44, 186, 2, 191, 88, 190, 14 |
| 253 | M | 181, 44, 186, 253, 191, 88, 190, 14 |

**5.7.3**

| Address | Tag | Hit/ Miss | Contents |
|---|---|---|---|
| 3 | 1 | M | 1 |
| 180 | 90 | M | 1, 90 |
| 43 | 21 | M | 1, 90, 21 |
| 2 | 1 | H | 1, 90, 21 |
| 191 | 95 | M | 1, 90, 21, 95 |
| 88 | 44 | M | 1, 90, 21, 95, 44 |
| 190 | 95 | H | 1, 90, 21, 95, 44 |
| 14 | 7 | M | 1, 90, 21, 95, 44, 7 |
| 181 | 90 | H | 1, 90, 21, 95, 44, 7 |
| 44 | 22 | M | 1, 90, 21, 95, 44, 7, 22 |
| 186 | 143 | M | 1, 90, 21, 95, 44, 7, 22, 143 |
| 253 | 126 | M | 1, 90, 126, 95, 44, 7, 22, 143 |

The final reference replaces tag 21 in the cache, since tags 1 and 90 had been reused at time=3 and time=8 while 21 hadn't been used since time=2.

Miss rate = 9/12 = 75%

This is the best possible miss rate, since there were no misses on any block that had been previously evicted from the cache. In fact, the only eviction was for tag 21, which is only referenced once.

**5.7.4** L1 only:

$.07 \times 100 = 7$ ns

CPI = 7 ns / .5 ns = 14

Direct mapped L2:

$.07 \times (12 + 0.035 \times 100) = 1.1$ ns

CPI = ceiling(1.1 ns/.5 ns) = 3

8-way set associated L2:

$.07 \times (28 + 0.015 \times 100) = 2.1$ ns

CPI = ceiling(2.1 ns / .5 ns) = 5

Doubled memory access time, L1 only:

$.07 \times 200 = 14$ ns

CPI = 14 ns / .5 ns = 28

Doubled memory access time, direct mapped L2:

$.07 \times (12 + 0.035 \times 200) = 1.3$ ns

CPI = ceiling(1.3 ns/.5 ns) = 3

Doubled memory access time, 8-way set associated L2:

$.07 \times (28 + 0.015 \times 200) = 2.2$ ns

CPI = ceiling(2.2 ns / .5 ns) = 5

Halved memory access time, L1 only:

$.07 \times 50 = 3.5$ ns

CPI = 3.5 ns / .5 ns = 7

Halved memory access time, direct mapped L2:

$.07 \times (12 + 0.035 \times 50) = 1.0$ ns

CPI = ceiling(1.1 ns/.5 ns) = 2

Halved memory access time, 8-way set associated L2:

$.07 \times (28 + 0.015 \times 50) = 2.1\,\text{ns}$

CPI = ceiling(2.1 ns / .5 ns) = 5

**5.7.5** $.07 \times (12 + 0.035 \times (50 + 0.013 \times 100)) = 1.0\,\text{ns}$

Adding the L3 cache does reduce the overall memory access time, which is the main advantage of having a L3 cache. The disadvantage is that the L3 cache takes real estate away from having other types of resources, such as functional units.

**5.7.6** Even if the miss rate of the L2 cache was 0, a 50 ns access time gives

AMAT = $.07 \times 50 = 3.5$ ns, which is greater than the 1.1 ns and 2.1 ns given by the on-chip L2 caches. As such, no size will achieve the performance goal.

## 5.8

### 5.8.1

| 1096 days | 26304 hours |
|---|---|

### 5.8.2

| 0.9990875912% |
|---|

**5.8.3** Availability approaches 1.0. With the emergence of inexpensive drives, having a nearly 0 replacement time *for hardware* is quite feasible. However, replacing file systems and other data can take significant time. Although a drive manufacturer will not include this time in their statistics, it is certainly a part of replacing a disk.

**5.8.4** MTTR becomes the dominant factor in determining availability. However, availability would be quite high if MTTF also grew measurably. If MTTF is 1000 times MTTR, it the specific value of MTTR is not significant.

## 5.9

**5.9.1** Need to find minimum p such that $2^p >= p + d + 1$ and then add one. Thus 9 total bits are needed for SEC/DED.

**5.9.2** The (72,64) code described in the chapter requires an overhead of 8/64=12.5% additional bits to tolerate the loss of any single bit within 72 bits, providing a protection rate of 1.4%. The (137,128) code from part a requires an overhead of 9/128=7.0% additional bits to tolerate the loss of any single bit within 137 bits, providing a protection rate of 0.73%. The cost/performance of both codes is as follows:

(72,64) code => 12.5/1.4 = 8.9

(136,128) code => 7.0/0.73 = 9.6

The (72,64) code has a better cost/performance ratio.

**5.9.3** Using the bit numbering from section 5.5, bit 8 is in error so the value would be corrected to 0x365.

**5.10** Instructors can change the disk latency, transfer rate and optimal page size for more variants. Refer to Jim Gray's paper on the five-minute rule ten years later.

**5.10.1** 32 KB

**5.10.2** Still 32 KB

**5.10.3** 64 KB. Because the disk bandwidth grows much faster than seek latency, future paging cost will be more close to constant, thus favoring larger pages.

**5.10.4** 1987/1997/2007: 205/267/308 seconds. (or roughly five minutes)

**5.10.5** 1987/1997/2007: 51/533/4935 seconds. (or 10 times longer for every 10 years).

**5.10.6** (1) DRAM cost/MB scaling trend dramatically slows down; or (2) disk $/access/sec dramatically increase. (2) is more likely to happen due to the emerging flash technology.

**5.11**

**5.11.1**

| Address | Virtual Page | TLB H/M | TLB | | |
|---|---|---|---|---|---|
| | | | Valid | Tag | Physical Page |
| 4669 | 1 | TLB miss PT hit PF | 1 | 11 | 12 |
| | | | 1 | 7 | 4 |
| | | | 1 | 3 | 6 |
| | | | 1 (last access 0) | 1 | 13 |
| 2227 | 0 | TLB miss PT hit | 1 (last access 1) | 0 | 5 |
| | | | 1 | 7 | 4 |
| | | | 1 | 3 | 6 |
| | | | 1 (last access 0) | 1 | 13 |
| 13916 | 3 | TLB hit | 1 (last access 1) | 0 | 5 |
| | | | 1 | 7 | 4 |
| | | | 1 (last access 2) | 3 | 6 |
| | | | 1 (last access 0) | 1 | 13 |
| 34587 | 8 | TLB miss PT hit PF | 1 (last access 1) | 0 | 5 |
| | | | 1 (last access 3) | 8 | 14 |
| | | | 1 (last access 2) | 3 | 6 |
| | | | 1 (last access 0) | 1 | 13 |
| 48870 | 11 | TLB miss PT hit | 1 (last access 1) | 0 | 5 |
| | | | 1 (last access 3) | 8 | 14 |
| | | | 1 (last access 2) | 3 | 6 |
| | | | 1 (last access 4) | 11 | 12 |
| 12608 | 3 | TLB hit | 1 (last access 1) | 0 | 5 |
| | | | 1 (last access 3) | 8 | 14 |
| | | | 1 (last access 5) | 3 | 6 |
| | | | 1 (last access 4) | 11 | 12 |
| 49225 | 12 | TLB miss PT miss | 1 (last access 6) | 12 | 15 |
| | | | 1 (last access 3) | 8 | 14 |
| | | | 1 (last access 5) | 3 | 6 |
| | | | 1 (last access 4) | 11 | 12 |

## 5.11.2

| Address | Virtual Page | TLB H/M | TLB | | |
|---|---|---|---|---|---|
| | | | Valid | Tag | Physical Page |
| 4669 | 0 | TLB miss PT hit | 1 | 11 | 12 |
| | | | 1 | 7 | 4 |
| | | | 1 | 3 | 6 |
| | | | 1 (last access 0) | 0 | 5 |
| 2227 | 0 | TLB hit | 1 | 11 | 12 |
| | | | 1 | 7 | 4 |
| | | | 1 | 3 | 6 |
| | | | 1 (last access 1) | 0 | 5 |
| 13916 | 0 | TLB hit | 1 | 11 | 12 |
| | | | 1 | 7 | 4 |
| | | | 1 | 3 | 6 |
| | | | 1 (last access 2) | 0 | 5 |
| 34587 | 2 | TLB miss PT hit PF | 1 (last access 3) | 2 | 13 |
| | | | 1 | 7 | 4 |
| | | | 1 | 3 | 6 |
| | | | 1 (last access 2) | 0 | 5 |
| 48870 | 2 | TLB hit | 1 (last access 4) | 2 | 13 |
| | | | 1 | 7 | 4 |
| | | | 1 | 3 | 6 |
| | | | 1 (last access 2) | 0 | 5 |
| 12608 | 0 | TLB hit | 1 (last access 4) | 2 | 13 |
| | | | 1 | 7 | 4 |
| | | | 1 | 3 | 6 |
| | | | 1 (last access 5) | 0 | 5 |
| 49225 | 3 | TLB hit | 1 (last access 4) | 2 | 13 |
| | | | 1 | 7 | 4 |
| | | | 1 (last axxess 6) | 3 | 6 |
| | | | 1 (last access 5) | 0 | 5 |

A larger page size reduces the TLB miss rate but can lead to higher fragmentation and lower utilization of the physical memory.

### 5.11.3 Two-way set associative

| Address | Virtual Page | Tag | Index | TLB H/M | TLB | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | Valid | Tag | Physical Page | Index |
| 4669 | 1 | 0 | 1 | TLB miss PT hit PF | 1 | 11 | 12 | 0 |
| | | | | | 1 | 7 | 4 | 1 |
| | | | | | 1 | 3 | 6 | 0 |
| | | | | | 1 (last access 0) | 0 | 13 | 1 |
| 2227 | 0 | 0 | 0 | TLB miss PT hit | 1 (last access 1) | 0 | 5 | 0 |
| | | | | | 1 | 7 | 4 | 1 |
| | | | | | 1 | 3 | 6 | 0 |
| | | | | | 1 (last access 0) | 0 | 13 | 1 |
| 13916 | 3 | 1 | 1 | TLB miss PT hit | 1 (last access 1) | 0 | 5 | 0 |
| | | | | | 1 (last access 2) | 1 | 6 | 1 |
| | | | | | 1 | 3 | 6 | 0 |
| | | | | | 1 (last access 0) | 1 | 13 | 1 |
| 34587 | 8 | 4 | 0 | TLB miss PT hit PF | 1 (last access 1) | 0 | 5 | 0 |
| | | | | | 1 (last access 2) | 1 | 6 | 1 |
| | | | | | 1 (last access 3) | 4 | 14 | 0 |
| | | | | | 1 (last access 0) | 1 | 13 | 1 |
| 48870 | 11 | 5 | 1 | TLB miss PT hit | 1 (last access 1) | 0 | 5 | 0 |
| | | | | | 1 (last access 2) | 1 | 6 | 1 |
| | | | | | 1 (last access 3) | 4 | 14 | 0 |
| | | | | | 1 (last access 4) | 5 | 12 | 1 |
| 12608 | 3 | 1 | 1 | TLB hit | 1 (last access 1) | 0 | 5 | 0 |
| | | | | | 1 (last access 5) | 1 | 6 | 1 |
| | | | | | 1 (last access 3) | 4 | 14 | 0 |
| | | | | | 1 (last access 4) | 5 | 12 | 1 |
| 49225 | 12 | 6 | 0 | TLB miss PT miss | 1 (last access 6) | 6 | 15 | 0 |
| | | | | | 1 (last access 5) | 1 | 6 | 1 |
| | | | | | 1 (last access 3) | 4 | 14 | 0 |
| | | | | | 1 (last access 4) | 5 | 12 | 1 |

Direct mapped

| Address | Virtual Page | Tag | Index | TLB H/M | TLB | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | Valid | Tag | Physical Page | Index |
| 4669 | 1 | 0 | 1 | TLB miss PT hit PF | 1 | 11 | 12 | 0 |
| | | | | | 1 | 0 | 13 | 1 |
| | | | | | 1 | 3 | 6 | 2 |
| | | | | | 0 | 4 | 9 | 3 |
| 2227 | 0 | 0 | 0 | TLB miss PT hit | 1 | 0 | 5 | 0 |
| | | | | | 1 | 0 | 13 | 1 |
| | | | | | 1 | 3 | 6 | 2 |
| | | | | | 0 | 4 | 9 | 3 |
| 13916 | 3 | 0 | 3 | TLB miss PT hit | 1 | 0 | 5 | 0 |
| | | | | | 1 | 0 | 13 | 1 |
| | | | | | 1 | 3 | 6 | 2 |
| | | | | | 1 | 0 | 6 | 3 |
| 34587 | 8 | 2 | 0 | TLB miss PT hit PF | 1 | 2 | 14 | 0 |
| | | | | | 1 | 0 | 13 | 1 |
| | | | | | 1 | 3 | 6 | 2 |
| | | | | | 1 | 0 | 6 | 3 |
| 48870 | 11 | 2 | 3 | TLB miss PT hit | 1 | 2 | 14 | 0 |
| | | | | | 1 | 0 | 13 | 1 |
| | | | | | 1 | 3 | 6 | 2 |
| | | | | | 1 | 2 | 12 | 3 |
| 12608 | 3 | 0 | 3 | TLB miss PT hit | 1 | 2 | 14 | 0 |
| | | | | | 1 | 0 | 13 | 1 |
| | | | | | 1 | 3 | 6 | 2 |
| | | | | | 1 | 0 | 6 | 3 |
| 49225 | 12 | 3 | 0 | TLB miss PT miss | 1 | 3 | 15 | 0 |
| | | | | | 1 | 0 | 13 | 1 |
| | | | | | 1 | 3 | 6 | 2 |
| | | | | | 1 | 0 | 6 | 3 |

All memory references must be cross referenced against the page table and the TLB allows this to be performed without accessing off-chip memory (in the common case). If there were no TLB, memory access time would increase significantly.

**5.11.4** Assumption: "half the memory available" means half of the 32-bit virtual address space for each running application.

The tag size is $32 - \log_2(8192) = 32 - 13 = 19$ bits. All five page tables would require $5 \times (2\text{^}19/2 \times 4)$ bytes $= 5$ MB.

**5.11.5** In the two-level approach, the 2^19 page table entries are divided into 256 segments that are allocated on demand. Each of the second-level tables contain $2\text{^}(19-8) = 2048$ entries, requiring $2048 \times 4 = 8$ KB each and covering $2048 \times 8$ KB $= 16$ MB (2^24) of the virtual address space.

If we assume that "half the memory" means 2^31 bytes, then the minimum amount of memory required for the second-level tables would be 5 × (2^31 / 2^24) * 8 KB = 5 MB. The first-level tables would require an additional 5 × 128 × 6 bytes = 3840 bytes.

The maximum amount would be if all segments were activated, requiring the use of all 256 segments in each application. This would require 5 × 256 × 8 KB = 10 MB for the second-level tables and 7680 bytes for the first-level tables.

**5.11.6** The page index consists of address bits 12 down to 0 so the LSB of the tag is address bit 13.

A 16 KB direct-mapped cache with 2-words per block would have 8-byte blocks and thus 16 KB / 8 bytes = 2048 blocks, and its index field would span address bits 13 down to 3 (11 bits to index, 1 bit word offset, 2 bit byte offset). As such, the tag LSB of the cache tag is address bit 14.

The designer would instead need to make the cache 2-way associative to increase its size to 16 KB.

## 5.12

**5.12.1** Worst case is 2^(43−12) entries, requiring 2^(43−12) × 4 bytes = 2 ^33 = 8 GB.

**5.12.2** With only two levels, the designer can select the size of each page table segment. In a multi-level scheme, reading a PTE requires an access to each level of the table.

**5.12.3** In an inverted page table, the number of PTEs can be reduced to the size of the hash table plus the cost of collisions. In this case, serving a TLB miss requires an extra reference to compare the tag or tags stored in the hash table.

**5.12.4** It would be invalid if it was paged out to disk.

**5.12.5** A write to page 30 would generate a TLB miss. Software-managed TLBs are faster in cases where the software can pre-fetch TLB entries.

**5.12.6** When an instruction writes to VA page 200, and interrupt would be generated because the page is marked as read only.

## 5.13

**5.13.1** 0 hits

**5.13.2** 1 hit

**5.13.3** 1 hits or fewer

**5.13.4** 1 hit. Any address sequence is fine so long as the number of hits are correct.

**5.13.5** The best block to evict is the one that will cause the fewest misses in the future. Unfortunately, a cache controller cannot know the future! Our best alternative is to make a good prediction.

**5.13.6** If you knew that an address had limited temporal locality and would conflict with another block in the cache, it could improve miss rate. On the other hand, you could worsen the miss rate by choosing poorly which addresses to cache.

## 5.14

**5.14.1** Shadow page table: (1) VM creates page table, hypervisor updates shadow table; (2) nothing; (3) hypervisor intercepts page fault, creates new mapping, and invalidates the old mapping in TLB; (4) VM notifies the hypervisor to invalidate the process's TLB entries. Nested page table: (1) VM creates new page table, hypervisor adds new mappings in PA to MA table. (2) Hardware walks both page tables to translate VA to MA; (3) VM and hypervisor update their page tables, hypervisor invalidates stale TLB entries; (4) same as shadow page table.

**5.14.2** Native: 4; NPT: 24 (instructors can change the levels of page table)

Native: L; NPT: L×(L+2)

**5.14.3** Shadow page table: page fault rate.

NPT: TLB miss rate.

**5.14.4** Shadow page table: 1.03

NPT: 1.04

**5.14.5** Combining multiple page table updates

**5.14.6** NPT caching (similar to TLB caching)

## 5.15

**5.15.1** CPI= 1.5 + 120/10000 × (15+175) = 3.78

If VMM performance impact doubles => CPI = 1.5 + 120/10000 × (15+350) =5.88

If VMM performance impact halves => CPI = 1.5 + 120/10000 × (15+87.5) =2.73

**5.15.2** Non-virtualized CPI = 1.5 + 30/10000 × 1100 = 4.80

Virtualized CPI = 1.5 + 120/10000 × (15+175) + 30/10000 × (1100+175) = 7.60

Virtualized CPI with half I/O= 1.5 + 120/10000 × (15+175) + 15/10000 × (1100+175) = 5.69

I/O traps usually often require long periods of execution time that can be performed in the guest O/S, with only a small portion of that time needing to be spent in the VMM. As such, the impact of virtualization is less for I/O bound applications.

**5.15.3** Virtual memory aims to provide each application with the illusion of the entire address space of the machine. Virtual machines aims to provide each operating system with the illusion of having the entire machine to its disposal. Thus they both serve very similar goals, and offer benefits such as increased security. Virtual memory can allow for many applications running in the same memory space to not have to manage keeping their memory separate.

**5.15.4** Emulating a different ISA requires specific handling of that ISA's API. Each ISA has specific behaviors that will happen upon instruction execution, interrupts, trapping to kernel mode, etc. that therefore must be emulated. This can require many more instructions to be executed to emulate each instruction than was originally necessary in the target ISA. This can cause a large performance impact and make it difficult to properly communicate with external devices. An emulated system can potentially run faster than on its native ISA if the emulated code can be dynamically examined and optimized. For example, if the underlying machine's ISA has a single instruction that can handle the execution of several of the emulated system's instructions, then potentially the number of instructions executed can be reduced. This is similar to the case with the recent Intel processors that do micro-op fusion, allowing several instructions to be handled by fewer instructions.

## 5.16

**5.16.1** The cache should be able to satisfy the request since it is otherwise idle when the write buffer is writing back to memory. If the cache is not able to satisfy hits while writing back from the write buffer, the cache will perform little or no better than the cache without the write buffer, since requests will still be serialized behind writebacks.

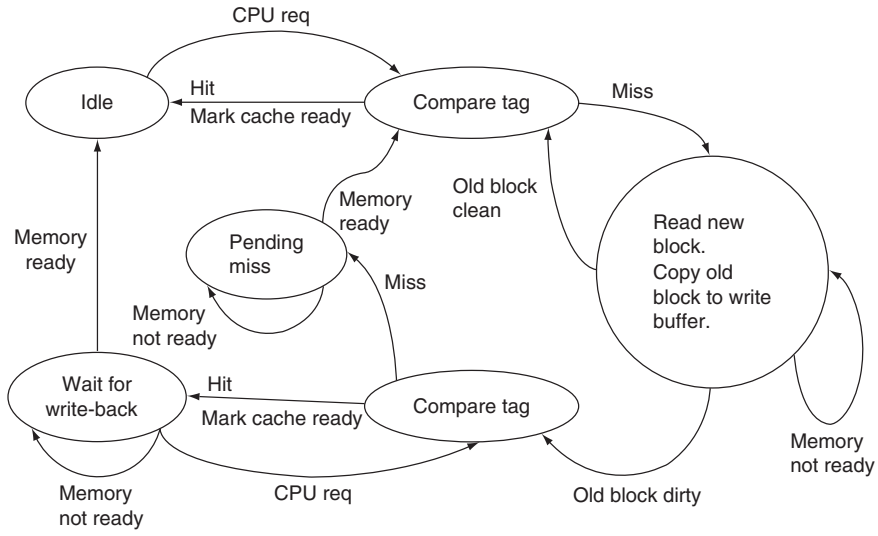
**5.16.2** Unfortunately, the cache will have to wait until the writeback is complete since the memory channel is occupied. Once the memory channel is free, the cache is able to issue the read request to satisfy the miss.

**5.16.3** Correct solutions should exhibit the following features:

1. The memory read should come before memory writes.

2. The cache should signal "Ready" to the processor before completing the write.

Example (simpler solutions exist; the state machine is somewhat underspecified in the chapter):




## 5.17

**5.17.1** There are 6 possible orderings for these instructions.

Ordering 1:

| P1 | P2 |
|---|---|
| X[0]++; | |
| X[1] = 3; | |
| | X[0]=5 |
| | X[1] += 2; |

Results: (5,5)

Ordering 2:

| P1 | P2 |
|---|---|
| X[0]++; | |
| | X[0]=5 |
| X[1] = 3; | |
| | X[1] += 2; |

Results: (5,5)

Ordering 3:

| P1 | P2 |
|---|---|
| | X[0]=5 |
| X[0]++; | |
| | X[1] += 2; |
| X[1] = 3; | |

Results: (6,3)

Ordering 4:

| P1 | P2 |
|---|---|
| X[0]++; | |
| | X[0]=5 |
| | X[1] += 2; |
| X[1] = 3; | |

Results: (5,3)

Ordering 5:

| P1 | P2 |
|---|---|
| | X[0]=5 |
| X[0]++; | |
| X[1] = 3; | |
| | X[1] += 2; |

Results: (6,5)

Ordering 6:

| P1 | P2 |
|---|---|
| | X[0]=5 |
| | X[1] += 2; |
| X[0]++; | |
| X[1] = 3; | |

(6,3)

If coherency isn't ensured:

P2's operations take precedence over P1's: (5,2)

### 5.17.2

| P1 | P1 cache status/ action | P2 | P2 cache status/action |
|---|---|---|---|
| | | X[0]=5 | invalidate X on other caches, read X in exclusive state, write X block in cache |
| | | X[1] += 2; | read and write X block in cache |
| X[0]++; | read value of X into cache | | X block enters shared state |
| | send invalidate message | | X block is invalided |
| | write X block in cache | | |
| X[1] = 3; | write X block in cache | | |

**5.17.3** Best case:

Orderings 1 and 6 above, which require only two total misses.

Worst case:

Orderings 2 and 3 above, which require 4 total cache misses.

**5.17.4** Ordering 1:

| P1 | P2 |
|---|---|
| A = 1 | |
| B = 2 | |
| A += 2; | |
| B++; | |
| | C = B |
| | D = A |

Result: (3,3)

Ordering 2:

| P1 | P2 |
|---|---|
| A = 1 | |
| B = 2 | |
| A += 2; | |
| | C = B |
| B++; | |
| | D = A |

Result: (2,3)

Ordering 3:

| P1 | P2 |
|---|---|
| A = 1 | |
| B = 2 | |
| | C = B |
| A += 2; | |
| B++; | |
| | D = A |

Result: (2,3)

Ordering 4:

| P1 | P2 |
|---|---|
| A = 1 | |
| | C = B |
| B = 2 | |
| A += 2; | |
| B++; | |
| | D = A |

Result: (0,3)

Ordering 5:

| P1 | P2 |
|---|---|
| | C = B |
| A = 1 | |
| B = 2 | |
| A += 2; | |
| B++; | |
| | D = A |

Result: (0,3)

Ordering 6:

| P1 | P2 |
|---|---|
| A = 1 | |
| B = 2 | |
| A += 2; | |
| | C = B |
| | D = A |
| B++; | |

Result: (2,3)

Ordering 7:

| P1 | P2 |
|---|---|
| A = 1 | |
| B = 2 | |
| | C = B |
| A += 2; | |
| | D = A |
| B++; | |

Result: (2,3)

Ordering 8:

| P1 | P2 |
|---|---|
| A = 1 | |
| | C = B |
| B = 2 | |
| A += 2; | |
| | D = A |
| B++; | |

Result: (0,3)

Ordering 9:

| P1 | P2 |
|---|---|
| | C = B |
| A = 1 | |
| B = 2 | |
| A += 2; | |
| | D = A |
| B++; | |

Result: (0,3)

Ordering 10:

| P1 | P2 |
|---|---|
| A = 1 | |
| B = 2 | |
| | C = B |
| | D = A |
| A += 2; | |
| B++; | |

Result: (2,1)

Ordering 11:

| P1 | P2 |
|---|---|
| A = 1 | |
| | C = B |
| B = 2 | |
| | D = A |
| A += 2; | |
| B++; | |

Result: (0,1)

Ordering 12:

| P1 | P2 |
|---|---|
| | C = B |
| A = 1 | |
| B = 2 | |
| | D = A |
| A += 2; | |
| B++; | |

Result: (0,1)

Ordering 13:

| P1 | P2 |
|---|---|
| A = 1 | |
| | C = B |
| | D = A |
| B = 2 | |
| A += 2; | |
| B++; | |

Result: (0,1)

Ordering 14:

| P1 | P2 |
|---|---|
| | C = B |
| A = 1 | |
| | D = A |
| B = 2 | |
| A += 2; | |
| B++; | |

Result: (0,1)

Ordering 15:

| P1 | P2 |
|---|---|
| | C = B |
| | D = A |
| A = 1 | |
| B = 2 | |
| A += 2; | |
| B++; | |

Result: (0,0)

**5.17.5** Assume B=0 is seen by P2 but not preceding A=1

Result: (2,0)

**5.17.6** Write back is simpler than write through, since it facilitates the use of exclusive access blocks and lowers the frequency of invalidates. It prevents the use of write-broadcasts, but this is a more complex protocol.

The allocation policy has little effect on the protocol.

## 5.18

**5.18.1** Benchmark A

$$AMAT_{private} = (1/32) \times 5 + 0.0030 \times 180 = 0.70$$

$$AMAT_{shared} = (1/32) \times 20 + 0.0012 \times 180 = 0.84$$

Benchmark B

$$AMAT_{private} = (1/32) \times 5 + 0.0006 \times 180 = 0.26$$

$$AMAT_{shared} = (1/32) \times 20 + 0.0003 \times 180 = 0.68$$

Private cache is superior for both benchmarks.

**5.18.2** Shared cache latency doubles for shared cache. Memory latency doubles for private cache.

Benchmark A

$$AMAT_{private} = (1/32) \times 5 + 0.0030 \times 360 = 1.24$$

$$AMAT_{shared} = (1/32) \times 40 + 0.0012 \times 180 = 1.47$$

Benchmark B

$$AMAT_{private} = (1/32) \times 5 + 0.0006 \times 360 = 0.37$$

$$AMAT_{shared} = (1/32) \times 40 + 0.0003 \times 180 = 1.30$$

Private is still superior for both benchmarks.

### 5.18.3

| | Shared L2 | Private L2 |
|---|---|---|
| **Single threaded** | No advantage. No disadvantage. | No advantage. No disadvantage. |
| **Multi-threaded** | Shared caches can perform better for workloads where threads are tightly coupled and frequently share data. | Threads often have private working sets, and using a private L2 prevents cache contamination and conflict misses between threads. |
| | No disadvantage. | |
| **Multiprogrammed** | No advantage except in rare cases where processes communicate. The disadvantage is higher cache latency. | Caches are kept private, isolating data between processes. This works especially well if the OS attempts to assign the same CPU to each process. |
| Having private L2 caches with a shared L3 cache is an effective compromise for many workloads, and this is the scheme used by many modern processors. | | |

**5.18.4** A non-blocking shared L2 cache would reduce the latency of the L2 cache by allowing hits for one CPU to be serviced while a miss is serviced for another CPU, or allow for misses from both CPUs to be serviced simultaneously. A non-blocking private L2 would reduce latency assuming that multiple memory instructions can be executed concurrently.

**5.18.5** 4 times.

**5.18.6** Additional DRAM bandwidth, dynamic memory schedulers, multi-banked memory systems, higher cache associativity, and additional levels of cache.

**f.** Processor: out-of-order execution, larger load/store queue, multiple hardware threads;

Caches: more miss status handling registers (MSHR)

Memory: memory controller to support multiple outstanding memory requests

## 5.19

**5.19.1** `srcIP` and `refTime` fields. 2 misses per entry.

**5.19.2** Group the `srcIP` and `refTime` fields into a separate array.

**5.19.3** `peak_hour (int status); // peak hours of a given status`

Group `srcIP`, `refTime` and status together.

**5.19.4** Answers will vary depending on which data set is used.

Conflict misses do not occur in fully associative caches.

Compulsory (cold) misses are not affected by associativity.

Capacity miss rate is computed by subtracting the compulsory miss rate and the fully associative miss rate (compulsory + capacity misses) from the total miss rate. Conflict miss rate is computed by subtracting the cold and the newly computed capacity miss rate from the total miss rate.

The values reported are miss rate per instruction, as opposed to miss rate per memory instruction.

**5.19.5** Answers will vary depending on which data set is used.

**5.19.6** apsi/mesa/ammp/mcf all have such examples.

Example cache: 4-block caches, direct-mapped vs. 2-way LRU.

Reference stream (blocks): 1 2 2 6 1.

# 6

# Solutions

**6.1** There is no single right answer for this question. The purpose is to get students to think about parallelism present in their daily lives. The answer should have at least 10 activities identified.

**6.1.1** Any reasonable answer is correct here.

**6.1.2** Any reasonable answer is correct here.

**6.1.3** Any reasonable answer is correct here.

**6.1.4** The student is asked to quantify the savings due to parallelism. The answer should consider the amount of overlap provided through parallelism and should be less than or equal to (if no parallelism was possible) to the original time computed if each activity was carried out serially.

## 6.2

**6.2.1** For this set of resources, we can pipeline the preparation. We assume that we do not have to reheat the oven for each cake.

Preheat Oven

Mix ingredients in bowl for Cake 1

Fill cake pan with contents of bowl and bake Cake 1. Mix ingredients for Cake 2 in bowl.

Finish baking Cake 1. Empty cake pan. Fill cake pan with bowl contents for Cake 2 and bake Cake 2. Mix ingredients in bowl for Cake 3.

Finish baking Cake 2. Empty cake pan. Fill cake pan with bowl contents for Cake 3 and bake Cake 3.

Finish baking Cake 3. Empty cake pan.

**6.2.2** Now we have 3 bowls, 3 cake pans and 3 mixers. We will name them A, B, and C.

Preheat Oven

Mix incredients in bowl A for Cake 1

Fill cake pan A with contents of bowl A and bake for Cake 1. Mix ingredients for

Cake 2 in bowl A.

Finish baking Cake 1. Empty cake pan A. Fill cake pan A with contents of bowl A for Cake 2. Mix ingredients in bowl A for Cake 3.

Finishing baking Cake 2. Empty cake pan A. Fill cake pan A with contents of bowl A for Cake 3.

Finish baking Cake 3. Empty cake pan A.

The point here is that we cannot carry out any of these items in parallel because we either have one person doing the work, or we have limited capacity in our oven.

**6.2.3** Each step can be done in parallel for each cake. The time to bake 1 cake, 2 cakes or 3 cakes is exactly the same.

**6.2.4** The loop computation is equivalent to the steps involved to make one cake. Given that we have multiple processors (or ovens and cooks), we can execute instructions (or cook multiple cakes) in parallel. The instructions in the loop (or cooking steps) may have some dependencies on prior instructions (or cooking steps) in the loop body (cooking a single cake).

Data-level parallelism occurs when loop iterations are independent (i.e., no loop carried dependencies).

Task-level parallelism includes any instructions that can be computed on parallel execution units, are similar to the independent operations involved in making multiple cakes.

## 6.3

**6.3.1** While binary search has very good serial performance, it is difficult to parallelize without modifying the code. So part A asks to compute the speedup factor, but increasing X beyond 2 or 3 should have no benefits. While we can perform the comparison of low and high on one core, the computation for mid on a second core, and the comparison for A[mid] on a third core, without some restructuring or speculative execution, we will not obtain any speedup. The answer should include a graph, showing that no speedup is obtained after the values of 1, 2, or 3 (this value depends somewhat on the assumption made) for Y.

**6.3.2** In this question, we suggest that we can increase the number of cores (to each the number of array elements). Again, given the current code, we really cannot obtain any benefit from these extra cores. But if we create threads to compare the N elements to the value X and perform these in parallel, then we can get ideal speedup (Y times speedup), and the comparison can be completed in the amount of time to perform a single comparison.

**6.4.** This problem illustrates that some computations can be done in parallel if serial code is restructured. But more importantly, we may want to provide for SIMD operations in our ISA, and allow for data-level parallelism when performing the same operation on multiple data items.

**6.4.1** This is a straightforward computation. The first instruction is executed once, and the loop body is executed 998 times.

Version 1—17,965 cycles

Version 2—22,955 cycles

Version 3—20,959 cycles

**6.4.2** Array elements D[j] and D[j−1] will have loop carried dependencies. These will `$f4` in the current iteration and `$f0` in the next iteration.

**6.4.3** This is a very challenging problem and there are many possible implementations for the solution. The preferred solution will try to utilize the two nodes by unrolling the loop 4 times (this already gives you a substantial speedup by eliminating many loop increment, branch and load instructions). The loop body running on node 1 would look something like this (the code is not the most efficient code sequence):

```
addiu $s1, $zero, 996
l.d $f0, -16($s0)
l.d $f2, -8($s0)
loop:
add.d $f4, $f2, $f0
add.d $f6, $f4, $f2
Send (2, $f4)
Send (2, $f6)
s.d $f4, 0($s0)
s.d $f6, 8($s0)
Receive($f8)
add.d $f10, $f8, $f6
add.d $f0, $f10, $f8
Send (2, $f10)
Send (2, $f0)
s.d. $f8, 16($s0)
s.d $f10, 24($s0)
s.d $f0 32($s0)
Receive($f2)
s.d $f2 40($s0)
addiu $s0, $s0, 48
bne $s0, $s1, loop
add.d $f4, $f2, $f0
add.d $f6, $f4, $f2
add.d $f10, $f8, $f6
s.d $f4, 0($s0)
s.d $f6, 8($s0)
s.d $f8, 16($s0)
```

The code on node 2 would look something like this:

```
 addiu $s2, $zero, 0
 loop:
Receive ($f12)
Receive ($f14)
add.d $f16, $f14, $f12
Send(1, $f16)
Receive ($f12)
Receive ($f14)
add.d $f16, $f14, $f12
Send(1, $f16)
Receive ($f12)
Receive ($f14)
add.d $f16, $f14, $f12
Send(1, $f16)
Receive ($f12)
Receive ($f14)
add.d $f16, $f14, $f12
Send(1, $f16)
addiu $s2, $s2, 1
bne $s2, 83, loop
```

Basically Node 1 would compute 4 adds each loop iteration, and Node 2 would compute 4 adds. The loop takes 1463 cycles, which is much better than close to 18K. But the unrolled loop would run faster given the current send instruction latency.

**6.4.4** The loop network would need to respond within a single cycle to obtain a speedup. This illustrates why using distributed message passing is difficult when loops contain loop-carried dependencies.

## 6.5

**6.5.1** This problem is again a divide and conquer problem, but utilizes recursion to produce a very compact piece of code. In part A the student is asked to compute the speedup when the number of cores is small. When forming the lists, we spawn a thread for the computation of left in the MergeSort code, and spawn a thread for the computation of the right. If we consider this recursively, for m initial elements in the array, we can utilize $1 + 2 + 4 + 8 + 16 + \ldots \log_2(m)$ processors to obtain speedup.

**6.5.2** In this question, $\log_2(m)$ is the largest value of Y for which we can obtain any speedup without restructuring. But if we had m cores, we could perform sorting using a very different algorithm. For instance, if we have greater than m/2 cores, we can compare all pairs of data elements, swap the elements if the left element is greater than the right element, and then repeat this step m times. So this is one possible answer for the question. It is known as parallel comparison sort. Various comparison sort algorithms include odd-even sort and cocktail sort.

## 6.6

**6.6.1** This problem presents an "embarrassingly parallel" computation and asks the student to find the speedup obtained on a 4-core system. The computations involved are: (m × p × n) multiplications and (m × p × (n − 1)) additions. The multiplications and additions associated with a single element in C are dependent (we cannot start summing up the results of the multiplications for an element until two products are available). So in this question, the speedup should be very close to 4.

**6.6.2** This question asks about how speedup is affected due to cache misses caused by the 4 cores all working on different matrix elements that map to the same cache line. Each update would incur the cost of a cache miss, and so will reduce the speedup obtained by a factor of 3 times the cost of servicing a cache miss.

**6.6.3** In this question, we are asked how to fix this problem. The easiest way to solve the false sharing problem is to compute the elements in C by traversing the matrix across columns instead of rows (i.e., using index-j instead of index-i). These elements will be mapped to different cache lines. Then we just need to make sure we process the matrix index that is computed ( i, j) and (i + 1, j) on the same core. This will eliminate false sharing.

## 6.7

**6.7.1** x = 2, y = 2, w = 1, z = 0

x = 2, y = 2, w = 3, z = 0

x = 2, y = 2, w = 5, z = 0

x = 2, y = 2, w = 1, z = 2

x = 2, y = 2, w = 3, z = 2

x = 2, y = 2, w = 5, z = 2

x = 2, y = 2, w = 1, z = 4

x = 2, y = 2, w = 3, z = 4

x = 3, y = 2, w = 5, z = 4

**6.7.2** We could set synchronization instructions after each operation so that all cores see the same value on all nodes.

## 6.8

**6.8.1** If every philosopher simultaneously picks up the left fork, then there will be no right fork to pick up. This will lead to starvation.

**6.8.2** The basic solution is that whenever a philosopher wants to eat, she checks both forks. If they are free, then she eats. Otherwise, she waits until a neighbor contacts her. Whenever a philosopher finishes eating, she checks to see if her neighbors want to eat and are waiting. If so, then she releases the fork to one of them and lets them eat. The difficulty is to first be able to obtain both forks without another philosopher interrupting the transition between checking and acquisition. We can implement this a number of ways, but a simple way is to accept requests for forks in a centralized queue, and give out forks based on the priority defined by being closest to the head of the queue. This provides both deadlock prevention and fairness.

**6.8.3** There are a number or right answers here, but basically showing a case where the request of the head of the queue does not have the closest forks available, though there are forks available for other philosophers.

**6.8.4** By periodically repeating the request, the request will move to the head of the queue. This only partially solves the problem unless you can guarantee that all philosophers eat for exactly the same amount of time, and can use this time to schedule the issuance of the repeated request.

## 6.9

### 6.9.1

| Core 1 | Core 2 |
|---|---|
| A3 | B1, B4 |
| A1, A2 | B1, B4 |
| A1, A4 | B2 |
| A1 | B3 |

### 6.9.2

| FU1 | FU2 |
|---|---|
| A1 | A2 |
| A1 | |
| A1 | |
| B1 | B2 |
| B1 | |
| A3 | |
| A4 | |
| B2 | |
| B4 | |

### 6.9.3

| FU1 | FU2 |
|---|---|
| A1 | B1 |
| A1 | B1 |
| A1 | B2 |
| A2 | B3 |
| A3 | B4 |
| A4 | |

**6.10** This is an open-ended question.

### 6.11

**6.11.1** The answer should include a MIPS program that includes 4 different processes that will compute ¼ of the sums. Assuming that memory latency is not an issue, the program should get linear speed when run on the 4 processors (there is no communication necessary between threads). If memory is being considered in the answer, then the array blocking should consider preserving spatial locality so that false sharing is not created.

**6.11.2** Since this program is highly data parallel and there are no data dependencies, a 8× speedup should be observed. In terms of instructions, the SIMD machine should have fewer instructions (though this will depend upon the SIMD extensions).

**6.12** This is an open-ended question that could have many possible answers. The key is that the student learns about MISD and compares it to an SIMD machine.

**6.13** This is an open-ended question that could have many answers. The key is that the students learn about warps.

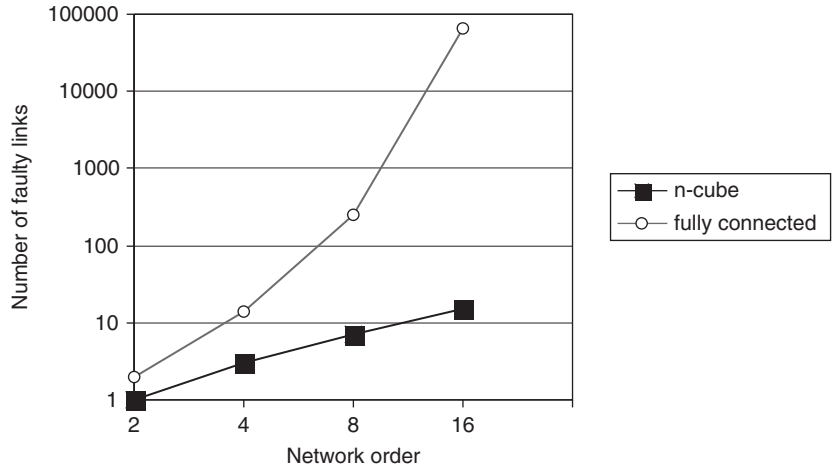
**6.14** This is an open-ended programming assignment. The code should be tested for correctness.

**6.15** This question will require the students to research on the Internet both the AMD Fusion architecture and the Intel QuickPath technology. The key is that students become aware of these technologies. The actual bandwidth and latency values should be available right off the company websites, and will change as the technology evolves.

### 6.16

**6.16.1** For an n-cube of order N ($2^N$ nodes), the interconnection network can sustain N−1 broken links and still guarantee that there is a path to all nodes in the network.

**6.16.2** The plot below shows the number of network links that can fail and still guarantee that the network is not disconnected.




## 6.17

**6.17.1** Major differences between these suites include:
Whetstone—designed for floating point performance specifically
PARSEC—these workloads are focused on multithreaded programs

**6.17.2** Only the PARSEC benchmarks should be impacted by sharing and synchronization. This should not be a factor in Whetstone.

## 6.18

**6.18.1** Any reasonable C program that performs the transformation should be accepted.

**6.18.2** The storage space should be equal to (R + R) times the size of a single precision floating point number + (m + 1) times the size of the index, where R is the number of non-zero elements and m is the number of rows. We will assume each floating-point number is 4 bytes, and each index is a short unsigned integer that is 2 bytes. For Matrix *X* this equals 111 bytes.

**6.18.3** The answer should include results for both a brute-force and a computation using the Yale Sparse Matrix Format.

**6.18.4** There are a number of more efficient formats, but their impact should be marginal for the small matrices used in this problem.

## 6.19

**6.19.1** This question presents three different CPU models to consider when executing the following code:

```
if (X[i][j] > Y[i][j])
    count++;
```

**6.19.2** There are a number of acceptable answers here, but they should consider the capabilities of each CPU and also its frequency. What follows is one possible answer:

Since X and Y are FP numbers, we should utilize the vector processor (CPU C) to issue 2 loads, 8 matrix elements in parallel from A and 8 matrix elements from B, into a single vector register and then perform a vector subtract. We would then issue 2 vector stores to put the result in memory.

Since the vector processor does not have comparison instructions, we would have CPU A perform 2 parallel conditional jumps based on floating point registers. We would increment two counts based on the conditional compare. Finally, we could just add the two counts for the entire matrix. We would not need to use core B.

**6.19.3** The point of the problem is to show that it is difficult to perform an operation on individual vector elements when utilizing a vector processor. What might be a nice instruction to add would be a vector comparison that would allow for us to compare two vectors and produce a scalar value of the number of elements where one vector was larger the other. This would reduce the computation to a single instruction for the comparison of 8 FP number pairs, and then an integer computation for summing up all of these values.

**6.20** This question looks at the amount of queuing that is occurring in the system given a maximum transaction processing rate, and the latency observed on average by a transaction. The latency includes both the service time (which is computed by the maximum rate) and the queue time.

**6.20.1** So for a max transaction processing rate of 5000/sec, and we have 4 cores contributing, we would see an average latency of .8 ms if there was no queuing taking place. Thus, each core must have 1.25 transactions either executing or in some amount of completion on average.

So the answers are:

| Latency | Max TP rate | Avg. # requests per core |
|---|---|---|
| 1 ms | 5000/sec | 1.25 |
| 2 ms | 5000/sec | 2.5 |
| 1 ms | 10,000/sec | 2.5 |
| 2 ms | 10,000/sec | 5 |

**6.20.2** We should be able to double the maximum transaction rate by doubling the number of cores.

**6.20.3** The reason this does not happen is due to memory contention on the shared memory system.