1

Solutions

1.1 **Personal computer:** Computer that emphasizes delivery of good performance to a single user at low cost and usually executes third-party software.

Server: Computer used for large workloads and usually accessed via a network.

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. Use Abstraction to Simplify Design

1.3 The program is compiled into an assembly language program, which is itself assembled into a machine-language program.

1.4

a. 1280×1024 pixels = 1,310,720 pixels => $1{,}310{,}720 \times 3 = 3{,}932{,}160$ 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

Desktop Processor	Year	Tech	Max. Clock Speed (GHz)	Integer IPC/core	Cores	Max. DRAM Bandwidth (GB/s)	SP Floating Point (Gflop/s)	L3 cache (MiB)
Westmere i7-620	2100	32	3.33	4	2	17.1	107	4
Ivy Bridge i7-3770K	2013	22	3.90	6	4	25.6	250	8
Broadwell i7-6700K	2015	14	4.20	8	4	34.1	269	8
Kaby Lake i7-7700K	2017	14	4.50	8	4	38.4	288	8
Coffee Lake i7-9700K	2019	14	4.90	8	8	42.7	627	12
Imp./year		20%	4%	7%	15%	10%	19%	12%
Doubles every		4 years	18 years	10 years	5 years	7 years	4 years	6 years

1.7

a. Class A: 10^5 instr. Class B: 2×10^5 instr. Class C: 5×10^5 instr. Class D: 2×10^5 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) = $10^5 \times 2 + 2 \times 10^5 \times 2 + 5 \times 10^5 \times 2 + 2 \times 10^5 \times 2 = 20 \times 10^5$

1.8

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

Compiler A $CPI = 1.1$

Compiler B $CPI = 1.25$

b. $f_B/f_A = (No.\ instr.(B) \times CPI(B))/(No.\ instr.(A) \times CPI(A)) = 1.37$

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

$T_B/T_{new} = 2.27$

1.9

1.9.1 $C = 2 \times DP/(V^2 \times F)$

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

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

1.9.2 Pentium 4: $10/100 = 10\%$

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

1.9.3 $(S_{new} + D_{new})/(S_{old} + D_{old}) = 0.90$

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

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

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

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:

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

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

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

$V_{new} = 0.64\,V$

p	# arith inst.	# L/S inst.	# branch inst.	cycles	ex. time	speedup
1	2.56E9	1.28E9	2.56E8	1.92E10	9.60	1.00
2	1.83E9	9.14E8	2.56E8	1.41E10	7.04	1.36
4	9.14E8	4.57E8	2.56E8	7.68E9	3.84	2.50
8	4.57E8	2.29E8	2.56E8	4.48E9	2.24	4.29

1.10

1.10.1

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

1.10.2

1.10.3 3

1.11

1.11.1 $\text{die area}_{15\text{cm}} = \text{wafer area/dies per wafer} = \pi \times 7.5^2/84 = 2.10\,\text{cm}^2$

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

$\text{die area}_{20\text{cm}} = \text{wafer area/dies per wafer} = \pi \times 10^2/100 = 3.14\,\text{cm}^2$

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

1.11.2 $\text{cost/die}_{15\text{cm}} = 12/(84 \times 0.9593) = 0.1489$

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

1.11.3 $\text{die area}_{15\text{cm}} = \text{wafer area/dies per wafer} = \pi \times 7.5^2/(84 \times 1.1) = 1.91\,\text{cm}^2$

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

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

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

1.11.4 $\text{defects per area}_{0.92} = (1-y^{.5})/(y^{.5} \times \text{die\_area}/2) = (1 - 0.92^{.5})/(0.92^{.5} \times 2/2) = 0.043\,\text{defects/cm}^2$

defects per $area_{0.95} = (1–y^{-.5})/(y^{-.5} \times die\_area/2) = (1 - 0.95^{-.5})/(0.95^{-.5} \times 2/2) = 0.026$ defects/cm^2

1.12

1.12.1 CPI = clock rate × CPU time/instr. count

clock rate = 1/cycle time = 3 GHz

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

1.12.2 SPEC ratio = ref. time/execution time

SPEC ratio(bzip2) = 9650/750 = 12.86

1.12.3 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 number of instructions, that is 10%.

1.12.4 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%.

1.12.5 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%.

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

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

1.12.7 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.

1.12.8 700/750 = 0.933. CPU time reduction: 6.7%

1.12.9 No. instr. = CPU time × clock rate/CPI

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

1.12.10 Clock rate = No. instr. × CPI/CPU time.

Clock rate_{new} = No. instr. × CPI/0.9 × CPU time = 1/0.9 clock rate_{old} = 3.33 GHz

1.12.11 Clock rate = No. instr. × CPI/CPU time.

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

1.13

1.13.1 $T(P1) = 5 \times 10^9 \times 0.9/(4 \times 10^9) = 1.125$ 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)

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

T(P1) = 2.25 3 1021 s

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

1.13.3 MIPS = Clock rate × 10^{-6}/CPI

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

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

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

1.13.4 MFLOPS = No. FP operations × 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)

1.14

1.14.1 $T_{fp} = 70 \times 0.8 = 56$ s. $T_{new} = 56 + 85 + 55 + 40 = 236$ s. Reduction: 5.6%

1.14.2 $T_{new} = 250 \times 0.8 = 200$ s, $T_{fp} + T_{l/s} + T_{branch} = 165$ s, $T_{int} = 35$ s. Reduction time INT: 58.8%

1.14.3 $T_{new} = 250 \times 0.8 = 200$ s, $T_{fp} + T_{int} + T_{l/s} = 210$ s. NO

1.15

1.15.1 Clock cycles = CPI_{fp} × No. FP instr. + CPI_{int} × No. INT instr. + $CPI_{l/s}$ × No. L/S instr. + CPI_{branch} × No. branch instr.

T_{CPU} = clock cycles/clock rate = clock cycles/2 × 10^9

clock cycles = 512 × 10^6; T_{CPU} = 0.256 s

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

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

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

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

1.15.2 Using the clock cycle data from a.

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

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

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

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

1.15.3 Clock cycles = CPI_{fp} × No. FP instr. + CPI_{int} × No. INT instr. + $CPI_{l/s}$ × No. L/S instr. + CPI_{branch} × No. branch instr.

T_{CPU} = clock cycles/clock rate = clock cycles/2 × 10^9

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

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

1.16

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

THIS PAGE INTENTIONALLY LEFT BLANK

2

Solutions

2.1
```
addi x5, x7,-5
add  x5, x5, x6
[addi f,h,-5 (note, no subi) add f,f,g]
```

2.2 `f = g+h+i`

2.3
```
sub     x30, x28, x29    // compute i-j
slli    x30, x30, 3      // multiply by 8 to convert the
                            word offset to a byte offset
add     x3, x30, x10
ld      x30, 0(x3)       // load A[i-j]
sd      x30, 64(x11)     // store in B[8]
```

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

```
slli    x30, x5, 3       // x30 = f*8
add     x30, x10, x30    // x30 = &A[f]
slli    x31, x6, 3       // x31 = g*8
add     x31, x11, x31    // x31 = &B[g]
ld      x5, 0(x30)       // f = A[f]
addi    x12, x30, 8      // x12 = &A[f]+8  (i.e. &A[f+1])
ld      x30, 0(x12)      // x30 = A[f+1]
add     x30, x30, x5     // x30 = A[f+1] + A[f]
sd      x30, 0(x31)      // B[g] = x30 (i.e. A[f+1] + A[f])
```

2.5

Little-Endian		Big-Endian	
Address	**Data**	**Address**	**Data**
3	ab	3	12
2	cd	2	ef
1	ef	1	cd
0	12	0	ab

2.6 `2882400018`

2.7
```
slli    x28, x28, 3      // x28 = i*8
add     x10,x10,x28      // x10 = &A[i]
ld      x28, 0(x10)      // x28 = A[i]
slli    x29, x29, 3      // x29 = j*8
add     x12,x11,x29      // x12 = &B[j]
ld      x29,0(x12)       // x29 = B[j]
add     x29, x28, x29    // x29 = B[i]+B[j]
sd      x29, 64(x11)     // B[8] = B[i]+B[j]
```

2.8
```
f = 2*(&A)
    addi   x30, x10, 8     // x30 = &A[1]
    addi   x31, x10, 0     // x31 = &A
    sd     x31, 0(x30)     // A[1] = &A
    ld     x30, 0(x30)     // x30 = A[1] = &A
    add    x5, x30, x31    // f = &A + &A = 2*(&A)
```

2.9

	type	opcode, funct3,7	rs1	rs2	rd	imm
`addi x30,x10,8`	I-type	0x13, 0x0, –	10	–	30	8
`addi x31,x10,0`	R-type	0x13, 0x0, –	10	–	31	0
`sd x31,0(x30)`	S-type	0x23, 0x3, –	31	30	–	0
`ld x30,0(x30)`	I-type	0x3, 0x3, –	30	–	30	0
`add x5, x30, x31`	R-type	0x33, 0x0, 0x0	30	31	5	–

2.10

2.10.1 `0x5000000000000000`

2.10.2 `overflow`

2.10.3 `0xB000000000000000`

2.10.4 `no overflow`

2.10.5 `0xD000000000000000`

2.10.6 `overflow`

2.11

2.11.1 There is an overflow if $128 + x6 > 2^{63} - 1$.
In other words, if $x6 > 2^{63} - 129$.
There is also an overflow if $128 + x6 < -2^{63}$.
In other words, if $x6 < -2^{63} - 128$ (which is impossible given the range of x6).

2.11.2 There is an overflow if $128 - x6 > 2^{63} - 1$.
In other words, if $x6 < -2^{63} + 129$.
There is also an overflow if $128 - x6 < -2^{63}$.
In other words, if $x6 > 2^{63} + 128$ (which is impossible given the range of x6).

2.11.3 There is an overflow if $x6 - 128 > 2^{63} - 1$.
In other words, if $x6 < 2^{63} + 127$ (which is impossible given the range of x6).
There is also an overflow if $x6 - 128 < -2^{63}$.
In other words, if $x6 < -2^{63} + 128$.

2.12 `R-type: add x1, x1, x1`

2.13 S-type: 0x25F3023 (0000 0010 0101 1111 0011 0000 0010 0011)

2.14 R-type: sub x6, x7, x5 (0x40538333: 0100 0000 0101 0011 1000 0011 0011 0011)

2.15 I-type: ld x3, 4(x27) (0x4DB183: 0000 0000 0100 1101 1011 0001 1000 0011)

2.16

2.16.1 The opcode would expand from 7 bits to 9.

The rs1, rs2, and rd fields would increase from 5 bits to 7 bits.

2.16.2 The opcode would expand from 7 bits to 12.

The rs1 and rd fields would increase from 5 bits to 7 bits. This change does not affect the imm field *per se*, but it might force the ISA designer to consider shortening the immediate field to avoid an increase in overall instruction size.

2.16.3 * Increasing the size of each bit field potentially makes each instruction longer, potentially increasing the code size overall.

* However, increasing the number of registers could lead to less register spillage, which would reduce the total number of instructions, possibly reducing the code size overall.

2.17

2.17.1 0x1234567ababefef8

2.17.2 0x2345678123456780

2.17.3 0x545

2.18 It can be done in eight RISC-V instructions:

```
addi x7, x0, 0x3f // Create bit mask for bits 16 to 11
slli x7, x7, 11   // Shift the masked bits
and  x28,x5, x7   // Apply the mask to x5
slli x7, x6, 15   // Shift the mask to cover bits 31
                     to 26
xori x7, x7, -1   // This is a NOT operation
and  x6, x6, x7   // "Zero out" positions 31 to
                     26 of x6
slli x28,x28,15   // Move selection from x5 into
                     positions 31 to 26
or   x6, x6, x28  // Load bits 31 to 26 from x28
```

2.19 xori x5, x6, -1

2.20
```
ld   x6, 0(x17)
slli x6, x6, 4
```

2.21 x6 = 2

2.22

2.22.1 [0x1ff00000, 0x200FFFFE]

2.22.2 [0x1FFFF000, 0x20000ffe]

2.23

2.23.1 The UJ instruction format would be most appropriate because it would allow the maximum number of bits possible for the "loop" parameter, thereby maximizing the utility of the instruction.

2.23.2 It can be done in three instructions:

```
loop:
     addi  x29, x29, -1   // Subtract 1 from x29
     bgt   x29, x0, loop  // Continue if x29 not
                             negative
     addi  x29, x29, 1    // Add back 1 that shouldn't
                             have been subtracted.
```

2.24

2.24.1 The final value of xs is 20.

2.24.2
```
acc = 0;
i = 10;
while (i ! = 0) {
   acc += 2;
   i--;
}
```

2.24.3 4*N + 1 instructions.

2.24.4 (Note: change condition ! = to > = in the while loop)

```
acc = 0;
i = 10;
while (i >= 0) {
   acc += 2;
   i--;
}
```

2.25 The C code can be implemented in RISC-V assembly as follows:

```
LOOPI:
      addi  x7,   x0, 0     // Init i = 0
      bge   x7,   x5, ENDI  // While i < a
      addi  x30,  x10, 0    // x30 = &D
      addi  x29,  x0, 0     // Init j = 0
LOOPJ:
      bge   x29,  x6, ENDJ  // While j < b
      add   x31,  x7, x29   // x31 = i+j
      sd    x31,  0(x30)    // D[4*j] = x31
      addi  x30,  x30, 32   // x30 = &D[4*(j+1)]
      addi  x29,  x29, 1    // j++
      jal   x0,   LOOPJ
ENDJ:
      addi  x7,   x7, 1     // i++;
      jal   x0,   LOOPI
ENDI:
```

2.26 The code requires 13 RISC-V instructions. When a = 10 and b = 1, this results in 123 instructions being executed.

2.27

```
// This C code corresponds most directly to the given
assembly.

int i;
for (i = 0; i < 100; i++) {
  result += *MemArray;
  MemArray++;
}
return result;

// However, many people would write the code this way:

int i;
for (i = 0; i < 100; i++) {
 result += MemArray[i];
}
return result;
```

2.28
```
      The address of the last element of MemArray can be
      used to terminate the loop:
      add x29, x10, 800      // x29 = &MemArray[101]
LOOP:
      ld    x7,   0(x10)
      add   x5,   x5, x7
      addi  x10,  x10, 8
      blt   x10,  x29, LOOP  // Loop until MemArray points
                                to one-past the last element
```

2.29
```
      // IMPORTANT! Stack pointer must reamin a multiple
      of 16!!!!

fib:
      beq    x10,  x0, done  // If n==0, return 0
      addi   x5,   x0, 1
      beq    x10,  x5, done  // If n==1, return 1
      addi   x2,   x2, -16   // Allocate 2 words of stack
                                space
      sd     x1,   0(x2)     // Save the return address
      sd     x10,  8(x2)     // Save the current n
      addi   x10,  x10, -1   // x10 = n-1
      jal    x1,   fib       // fib(n-1)
      ld     x5,   8(x2)     // Load old n from the stack
      sd     x10,  8(x2)     // Push fib(n-1) onto the stack
      addi   x10,  x5, -2    // x10 = n-2
      jal    x1,   fib       // Call fib(n-2)
      ld     x5,   8(x2)     // x5 = fib(n-1)
      add    x10,  x10, x5   // x10 = fib(n-1)+fib(n-2)
// Clean up:
      ld     x1,   0(x2)     // Load saved return address
      addi   x2,   x2, 16    // Pop two words from the stack
done:
      jalr   x0,   x1
```

2.30 [answers will vary]

2.31

```
// IMPORTANT! Stack pointer must remain a multiple of 16!!!
f:
  addi  x2, x2, -16   // Allocate stack space for 2 words
  sd    x1, 0(x2)     // Save return address
  add   x5, x12, x13  // x5 = c+d
  sd    x5, 8(x2)     // Save c+d on the stack
  jal   x1, g         // Call x10 = g(a,b)
  ld    x11, 8(x2)    // Reload x11= c+d from the stack
  jal   x1, g         // Call x10 = g(g(a,b), c+d)
  ld    x1, 0(x2)     // Restore return address
  addi  x2, x2, 16    // Restore stack pointer
  jalr  x0, x1
```

2.32 We can use the tail-call optimization for the second call to `g`, saving one instruction:

```
// IMPORTANT! Stack pointer must remain a multiple of 16!!!
f:
  addi  x2, x2, -16   // Allocate stack space for 2 words
  sd    x1, 0(x2)     // Save return address
  add   x5, x12, x13  // x5 = c+d
  sd    x5, 8(x2)     // Save c+d on the stack
  jal   x1, g         // Call x10 = g(a,b)
  ld    x11, 8(x2)    // Reload x11 = c+d from the stack
  ld    x1, 0(x2)     // Restore return address
  addi  x2, x2, 16    // Restore stack pointer
  jal   x0, g         // Call x10 = g(g(a,b), c+d)
```

2.33 *We have no idea what the contents of `x10-x14` are, g can set them as it pleases.

*We don't know what the precise contents of `x8` and `sp` are; but we do know that they are identical to the contents when `f` was called.

*Similarly, we don't know what the precise contents of `x1` are; but, we do know that it is equal to the return address set by the "`jal x1, f`" instruction that invoked `f`.

2.34

```
a_to_i:
  addi   x28,  x0, 10     # Just stores the constant 10
  addi   x29,  x0, 0      # Stores the running total
  addi   x5,   x0, 1      # Tracks whether input is positive
                            or negative
  # Test for initial '+' or '-'
  lbu    x6, 0(x10)       # Load the first character
  addi   x7, x0, 45       # ASCII '-'
  bne    x6, x7, noneg
  addi   x5, x0, -1       # Set that input was negative
  addi   x10, x10, 1      # str++
  jal    x0, main_atoi_loop

noneg:
  addi   x7, x0, 43       # ASCII '+'
  bne    x6, x7, main_atoi_loop
  addi   x10, x10, 1      # str++

main_atoi_loop:
  lbu    x6, 0(x10)       # Load the next digit
  beq    x6, x0, done     # Make sure next char is a digit,
                            or fail
  addi   x7, x0, 48       # ASCII '0'
  sub    x6, x6, x7
  blt    x6, x0, fail     # *str < '0'
  bge    x6, x28, fail    # *str >= '9'

  # Next char is a digit, so accumulate it into x29

  mul    x29, x29, x28    # x29 *= 10
  add    x29, x29, x6     # x29 += *str - '0'
  addi   x10, x10, 1      # str++
  jal    x0, main_atoi_loop

done:
  addi   x10, x29, 0      # Use x29 as output value
  mul    x10, x10, x5     # Multiply by sign
  jalr   x0, x1           # Return result

fail:
  addi   x10, x0, -1
  jalr   x0, x1
```

2.35

2.35.1 0x11

2.35.2 0x88

2.36
```
lui   x10,  0x11223
addi  x10,  x10, 0x344
slli  x10,  x10, 32
lui   x5,   0x55667
addi  x5,   x5, 0x788
add   x10,  x10, x5
```

2.37
```
setmax:
    try:
    lr.d  x5,  (x10)         # Load-reserve *shvar
    bge   x5,  x11, release  # Skip update if *shvar > x
    addi  x5,  x11, 0

release:
    sc.d  x7,  x5, (x10)
    bne   x7,  x0, try       # If store-conditional failed,
                               try again
    jalr  x0,  x1
```

2.38 When processors A and B begin executing this routine at the same time, at most one of them will execute the store-conditional instruction successfully, while the other will be forced to retry the operation. If processor A's store-conditional succeeds initially, then B will reenter the try block, and it will see the new value of shvar written by A when it finally succeeds. The hardware guarantees that both processors will eventually execute the code completely.

2.39

2.39.1 No. The resulting machine would be slower overall.
Current CPU requires (num arithmetic * 1 cycle) + (num load/store * 10 cycles) + (num branch/jump * 3 cycles) = 500*1 + 300*10 + 100*3 = 3800 cycles.
The new CPU requires (.75*num arithmetic * 1 cycle) + (num load/store * 10 cycles) + (num branch/jump * 3 cycles) = 375*1 + 300*10 + 100*3 = 3675 cycles.
However, given that each of the new CPU's cycles is 10% longer than the original CPU's cycles, the new CPU's 3675 cycles will take as long as 4042.5 cycles on the original CPU.

2.39.2 If we double the performance of arithmetic instructions by reducing their CPI to 0.5, then the the CPU will run the reference program in (500*.5) + (300*10) + 100*3 = 3550 cycles. This represents a speedup of 1.07.

If we improve the performance of arithmetic instructions by a factor of 10 (reducing their CPI to 0.1), then the the CPU will run the reference program in (500*.1) + (300*10) + 100*3 = 3350 cycles. This represents a speedup of 1.13.

2.40

2.40.1 Take the weighted average: 0.7*2 + 0.1*6 + 0.2*3 = 2.6.

2.40.2 For a 25% improvement, we must reduce the CPU to 2.6*.75 = 1.95. Thus, we want 0.7*x + 0.1*6 + 0.2*3 < = 1.95. Solving for x shows that the arithmetic instructions must have a CPI of at most 1.07.

2.40.3 For a 50% improvement, we must reduce the CPU to 2.6*.5 = 1.3. Thus, we want 0.7*x + 0.1*6 + 0.2*3 < = 1.3. Solving for x shows that the arithmetic instructions must have a CPI of at most 0.14.

2.41

```
ldr   x28, x5(x10), 3      // Load x28=A[f]
addi  x5, x5, 1            // f++
ldr   x29, x5(x10), 3      // Load x29=A[f+1]
add   x29, x29, x28        // Add x29 = A[f] + A[f+1]
sdr   x12, x6(x11), 3      // Store B[g] = x29
```

2.42

```
ldr   x28, x28, (x10), 3   // Load x28=A[i]
ldr   x29, x29, (x11), 3   // Load x29=B[j]
add   x29, x28, x29
sd    x29, 64(x11)         // Store B[8]=x29 (don't
                              need scaled store here)
```

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, by definition, are 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 one cycle to do the add, one cycle to do the shift, and one 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 one cycle to decide what to add, one cycle to do the add, one cycle to do each shift, and one 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. An 8 bit wide word requires seven adders in three levels. 3 × 4tu = 12 time units.

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

(Could also use 0x55, which is 64 + 16 + 4 + 1, and shift 0x33 left six times, add to it 0x33 shifted left four times, add to that 0x33 shifted left two 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 = 1.0×2^{-103}

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

normalize, move binary point 5 to the left,

1.1111101×2^5

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,

1.1111101×2^5

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 two to the left,

$0.3F40 \times 16^2$

sign = positive, exp = 64 + 2

Final bit pattern: 01000010001111110100000000000000

3.26 $-1.5625 \times 10^{-1} = -0.15625 \times 10^0$

$= -0.00101 \times 2^0$

move the binary point 2 to the right,

$= -0.101 \times 2^{-2}$

exponent = −2, fraction = −0.10100000000000000000000

answer: 11111111111010110000000000000000000

3.27 $-1.5625 \times 10^{-1} = -0.15625 \times 10^0$

$= -0.00101 \times 2^0$

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

exponent = −3 = −3 + 15 = 12, fraction = −0.0100000000

answer: 1011000100000000

3.28 $-1.5625 \times 10^{-1} = -0.15625 \times 10^0$

$= -0.00101 \times 2^0$

move the binary point 2 to the right,

$= -0.101 \times 2^{-2}$

exponent = −2, fraction = −0.10100000000000000000000000

answer: 10110000000000000000000000000101

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

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

$4.150390625 \times 10^{-1} = 0.4150390625 = 0.011010100111 = 1.1010100111 \times 2^{-2}$

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
1.01110011000001001110
```

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

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

$-8.0546875 \times -0.179931640625 = 1.4492931365966796875$

Some information was lost because the result did not fit into the available 10-bit field. Answer (only) off by 0.0000743865966796875.

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 0.00480769230.

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) + $1.1011101011 \times 2^{10}$ = 0110101011101011 = 1771

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

Exact: 0.398437 + 0.34375 + 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
1.1100000011 10 100000000 Guard 5 1, Round 5 0, Sticky
5 1: Round
```

$B \times C \quad 1.1100000100 \times 2^{-2}$

Exp: $-9-2 = -11$

Signs: both positive, result positive

Fraction:

```
(A)     1.1100000000
(B × C)× 1.1100000100
          -------------
           11100000000
     11100000000
    11100000000
   11100000000
  ---------------------
11.000100001110000000000    Normalize, add 1 to exponent
 1.1000100011 10 0000000000 Guard=1, Round=0, Sticky=0:
Round to even
```

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

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
```

$A \times (B + C)$ 1.1010101010×2^4

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
            --------------------------------
            10.00000010011000011111 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
                            --------------------------
                            10.00000001111101110100
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 value of the signals is as follows:

RegWrite	ALUSrc	ALUoperation	MemWrite	MemRead	MemToReg
true	0	"and"	false	false	0

Mathematically, the `MemRead` control wire is a "don't care": the instruction will run correctly regardless of the chosen value. Practically, however, `MemRead` should be set to false to prevent causing a segment fault or cache miss.

4.1.2 Registers, ALUsrc mux, ALU, and the MemToReg mux.

4.1.3 All blocks produce some output. The outputs of DataMemory and Imm Gen are not used.

4.2 `MemToReg` for `sd` and `beq`: Neither `sd` nor `beq` write a value to the register file. It doesn't matter which value the MemToReg mux passes to the register file because the register file ignores that value.

4.3

4.3.1 25 + 10 = 35%. Only `Load` and `Store` use Data memory.

4.3.2 100% Every instruction must be fetched from instruction memory before it can be executed.

4.3.3 28 + 25 + 10 + 11 + 2 = 76%. Only R-type instructions do not use the Sign extender.

4.3.4 The sign extend produces an output during every cycle. If its output is not needed, it is simply ignored.

4.4

4.4.1 Only `loads` are broken. `MemToReg` is either 1 or "don't care" for all other instructions.

4.4.2 I-type, `loads`, `stores` are all broken.

4.5 For context: The encoded instruction is `sd x12, 20(x13)`

4.5.1

ALUOp	ALU Control Lines
00	0010

4.5.2 The new PC is the old PC + 4. This signal goes from the PC, through the "PC + 4" adder, through the "branch" mux, and back to the PC.

4.5.3 `ALUsrc`: Inputs: `Reg[x12]` and `0x0000000000000014`; Output: `0x0000000000000014`

`MemToReg`: Inputs: `Reg[x13] + 0x14` and `<undefined>`; output: `<undefined>`

`Branch`: Inputs: `PC+4` and `PC + 0x28`

4.5.4 ALU inputs: `Reg[x13]` and `0x0000000000000014`

PC + 4 adder inputs: `PC` and `4`

Branch adder inputs: `PC` and `0x0000000000000028`

4.5.5 Read register 1 = `0x13` (base address)

Read register 2 = `0x12` (data to be stored)

Write register = `0x0` or don't-care (should not write back)

Write data = don't-care (should not write back)

RegWrite = false (should not write back)

4.6

4.6.1 No additional logic blocks are needed.

4.6.2
`Branch: false`
`MemRead: false` (See footnote from solution to problem 4.1.1.)
`MemToReg: 0`
`ALUop: 10` (or simply saying "add" is sufficient for this problem)
`MemWrite: false`
`ALUsrc: 1`
`RegWrite: 1`

4.7

4.7.1 `R-type`: 30 + 250 + 150 + 25 + 200 + 25 + 20 = 700ps

4.7.2 `ld`: 30 + 250 + 150 + 25 + 200 + 250 + 25 + 20 = 950ps

4.7.3 `sd`: 30 + 250 + 150 + 200 + 25 + 250 = 905

4.7.4 `beq`: 30 + 250 + 150 + 25 + 200 + 5 + 25 + 20 = 705

4.7.5 `I-type`: 30 + 250 + 150 + 25 + 200 + 25 + 20 = 700ps

4.7.6 950ps

4.8 Using the results from Problem 4.7, we see that the average time per instruction is

.52*700 + .25*950 + .11*905 + .12 * 705 = 785.6ps.

In contrast, a single-cycle CPU with a "normal" clock would require a clock cycle time of 950.

Thus, the speedup would be 925/787.6 = 1.174.

4.9

4.9.1 Without improvement: 950; With improvement: 1250

4.9.2 The running time of a program on the original CPU is 950*n. The running time on the improved CPU is 1250*(0.95)*n = 1187.5. Thus, the "speedup" is 0.8. (Thus, this "improved" CPU is actually slower than the original).

4.9.3 Because adding a multiply instruction will remove 5% of the instructions, the cycle time can grow to as much as 950/(0.95) = 1000. Thus, the time for the ALU can increase by up to 50 (from 200 to 250).

4.10

4.10.1 The additional registers will allow us to remove 12% of the loads and stores, or (0.12)*(0.25 + 0.1) = 4.2% of all instructions. Thus, the time to run n instructions will decrease from 950*n to 960*.958*n = 919.68*n. That corresponds to a speedup of 950/919.68 = 1.03.

4.10.2 The cost of the original CPU is 4496; the cost of the improved CPU is 4696.

PC: 5
I-Mem: 1000
Register file: 200
ALU: 100
D-Mem: 2000
Sign Extend: 100
Controls: 1000
adders: 30*2
muxes: 3*10
single gates: 1*1

Thus, for a 3% increase in performance, the cost of the CPU increases by about 4.4%.

4.10.3 From a strictly mathematical standpoint, it does not make sense to add more registers because the new CPU costs more per unit of performance. However, that simple calculation does not account for the utility of the performance. For example, in a real-time system, a 3% performance may make the difference between meeting or missing deadlines. In which case, the improvement would be well worth the 4.4% additional cost.

4.11

4.11.1 No new functional blocks are needed.

4.11.2 Only the control unit needs modification.

4.11.3 No new data paths are needed.

4.11.4 No new signals are needed.

4.12

4.12.1 No new functional blocks are needed.

4.12.2 The register file needs to be modified so that it can write to two registers in the same cycle. The ALU would also need to be modified to allow read data 1 or 2 to be passed through to write data 1.

4.12.3 The answer depends on the answer given in 4.12.2: Whichever input was not allowed to pass through the ALU above must now have a data path to write data 2.

4.12.4 There would need to be a second RegWrite control wire.

4.12.5 Many possible solutions.

4.13

4.13.1 We need some additional muxes to drive the data paths discussed in 4.13.3.

4.13.2 No functional blocks need to be modified.

4.13.3 There needs to be a path from the ALU output to data memory's write data port. There also needs to be a path from read data 2 directly to Data memory's Address input.

4.13.4 These new data paths will need to be driven by muxes. These muxes will require control wires for the selector.

4.13.5 Many possible solutions.

4.14 None: all instructions that use sign extend also use the register file, which is slower.

4.15

4.15.1 The new clock cycle time would be 750. ALU and Data Memory will now run in parallel, so we have effectively removed the faster of the two (the ALU with time 200) from the critical path.

4.15.2 Slower. The original CPU takes 950*n picoseconds to run n instructions. The same program will have approximately 1.35*n instructions when compiled for the new machine. Thus, the time on the new machine will be 750*1.35n = 1012.5*n. This represents a "speedup" of .93.

4.15.3 The number of loads and stores is the primary factor. How the loads and stores are used can also have an effect. For example, a program whose loads and stores tend to be to only a few different address may also run faster on the new machine.

4.15.4 This answer is a matter of opinion.

4.16

4.16.1 Pipelined: 350; non-pipelined: 1250

4.16.2 Pipelined: 1750; non-pipelined: 1250

4.16.3 Split the ID stage. This reduces the clock-cycle time to 300ps.

4.16.4 35%

4.16.5 65%

4.17 $n + k - 1$. Let's look at when each instruction is in the WB stage. In a k-stage pipeline, the 1st instruction doesn't enter the WB stage until cycle k. From that point on, at most one of the remaining $n - 1$ instructions is in the WB stage during every cycle.

This gives us a minimum of $k + (n - 1) = n + k - 1$ cycles.

4.18 x13 = 33 and x14 = 36

4.19 x15 = 54 (The code will run correctly because the result of the first instruction is written back to the register file at the beginning of the 5th cycle, whereas the final instruction reads the updated value of x1 during the second half of this cycle.)

4.20
```
addi x11, x12, 5
NOP
NOP
add x13, x11, x12
addi x14, x11, 15
NOP
add x15, x13, x12
```

4.21

4.21.1 The pipeline without forwarding requires 1.4*n*250ps. The pipeline with forwarding requires 1.05*n*300ps. The speedup is therefore (1.4*250)/(1.05*300) = 1.11.

4.21.2 Our goal is for the pipeline with forwarding to be faster than the pipeline without forwarding. Let y be the number of stalls remaining as a percentage of "code" instructions. Our goal is for 300*(1+y)*n < 250*1.4*n. Thus, y must be less than 16.7%.

4.21.3 This time, our goal is for 300(1 + y)*n < 250(1 + x)*n. This happens when y < (250x − 50)/300.

4.21.4 It cannot. In the best case, where forwarding eliminates the need for every NOP, the program will take time 300*n to run on the pipeline with forwarding. This is slower than the 250*1.075*n required on the pipeline with no forwarding.

4.21.5 Speedup is not possible when the solution to 4.21.3 is less than 0. Solving 0< (250x − 50)/300 for x gives that x must be at least 0.2.

4.22

4.22.1 Stalls are marked with **:

```
sd  x29, 12(x16)  IF ID EX ME WB
ld  x29, 8(x16)      IF ID EX ME WB
sub x17, x15, x14       IF ID EX ME WB
bez x17, label             ** ** IF ID EX ME WB
add x15, x11, x14                    IF ID EX ME WB
sub x15,x30,x14                         IF ID EX ME WB
```

4.22.2 Reordering code won't help. Every instruction must be fetched; thus, every data access causes a stall. Reordering code will just change the pair of instructions that are in conflict.

4.22.3 You can't solve this structural hazard with NOPs, because even the NOPs must be fetched from instruction memory.

4.22.4 35%. Every data access will cause a stall.

4.23

4.23.1 The clock period won't change because we aren't making any changes to the slowest stage.

4.23.2 Moving the MEM stage in parallel with the EX stage will eliminate the need for a cycle between loads and operations that use the result of the loads. This can potentially reduce the number of stalls in a program.

4.23.3 Removing the offset from `ld` and `sd` may increase the total number of instructions because some `ld` and `sd` instructions will need to be replaced with a `addi/ld` or `addi/sd` pair.

4.24 The second one. A careful examination of Figure 4.59 shows that the need for a stall is detected during the ID stage. It is this stage that prevents the fetch of a new instruction, effectively causing the add to repeat its ID stage.

4.25

4.25.1 … indicates a stall. The `!` indicates a stage that does not do useful work.

```
ld x10, 0(x13)      IF ID EX ME | WB
ld x11, 8(x13)         IF ID EX | ME WB
add x12, x10, x11         IF ID | .. EX ME! WB
addi x13, x13, -16           IF | .. ID EX  ME! WB
bnez x12, LOOP                  | .. IF ID  EX  ME! WB!
ld x10, 0(x13)                        IF  ID  EX  ME  WB
ld x11, 8(x13)                            IF  ID  EX  ME WB
add x12, x10, x11                             IF  ID  .. EX | ME! WB
addi x13, x13, -16                                IF  .. ID | EX  ME! WB
bnez x12, LOOP                                        IF | ID  EX  ME! WB!
Completely busy                 | N  N  N   N   N   N   N  N  |
```

4.25.2 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. As the diagram above shows, there are not any cycles during which every pipeline stage is doing useful work.

4.26

4.26.1
```
// EX to 1st only:
add x11, x12, x13
add x14, x11, x15
add x5, x6, x7

// MEM to 1st only:
ld x11, 0(x12)
add x15, x11, x13
add x5, x6, x7

// EX to 2nd only:
add x11, x12, x13
add x5, x6, x7
add x14, x11, x12

// MEM to 2nd only:
ld x11, 0(x12)
add x5, x6, x7
add x14, x11, x13

// EX to 1st and EX to 2nd:
add x11, x12, x13
add x5, x11, x15
add x16, x11, x12
```

4.26.2
```
// EX to 1st only: 2 nops
add x11, x12, x13
nop
nop
add x14, x11, x15
add x5, x6, x7

// MEM to 1st only: 2 stalls
ld x11, 0(x12)
nop
nop
add x15, x11, x13
add x5, x6, x7

// EX to 2nd only: 1 nop
add x11, x12, x13
add x5, x6, x7
nop
add x14, x11, x12
```

```
// MEM to 2nd only: 1 nop
ld x11, 0(x12)
add x5, x6, x7
nop
add x14, x11, x13

// EX to 1st and EX to 2nd: 2 nops
add x11, x12, x13
nop
nop
add x5, x11, x15
add x16, x11, x12
```

4.26.3 Consider this code:

```
ld  x11, 0(x5)       # MEM to 2nd --- one stall
add x12, x6, x7      # EX to 1st  --- two stalls
add x13, x11, x12
add x28, x29, x30
```

If we analyze each instruction separately, we would calculate that we need to add 3 stalls (one for a "MEM to 2nd" and two for an "EX to 1st only." However, as we can see below, we need only two stalls:

```
ld x11, 0(x5)
add  x12, x6, x7
nop
nop
add  x13, x11, x12
add  x28, x29, x30
```

4.26.4 Taking a weighted average of the answers from 4.26.2 gives 0.05*2 + 0.2*2 + 0.05*1 + 0.1*1 + 0.1*2 = 0.85 stalls per instruction (on average) for a CPI of 1.85. This means that 0.85/1.85 cycles, or 46%, are stalls.

4.26.5 The only dependency that cannot be handled by forwarding is from the MEM stage to the next instruction. Thus, 20% of instructions will generate one stall for a CPI of 1.2. This means that 0.2 out of 1.2 cycles, or 17%, are stalls.

4.26.6 If we forward from the EX/MEM register only, we have the following stalls/NOPs:

EX to 1st:	0
MEM to 1st:	2
EX to 2nd:	1
MEM to 2nd:	1
EX to 1st and 2nd:	1

This represents an average of 0.05*0 + 0.2*2 + 0.05*1 + 0.10*1 + 0.10*1 = 0.65 stalls/instruction. Thus, the CPI is 1.65.

IF we forward from MEM/WB only, we have the following stalls/NOPs:

EX to 1st:	1
MEM to 1st:	1
EX to 2nd:	0
MEM to 2nd:	0
EX to 1st and 2nd:	1

This represents an average of 0.05*1 + 0.2*1 + 0.1*1 = 0.35 stalls/instruction. Thus, the CPI is 1.35.

4.26.7

	No forwarding	EX/MEM	MEM/WB	Full Forwarding
CPI	1.85	1.65	1.35	1.2
Period	120	120	1.20	130
Time	222n	198n	162n	156n
Speedup	–	1.12	1.37	1.42

4.26.8 CPI for full forwarding is 1.2
CPI for "time travel" forwarding is 1.0
clock period for full forwarding is 130
clock period for "time travel" forwarding is 230

Speedup = (1.2*130)/ (1*230) = 0.68 (That means that "time travel" forwarding actually slows the CPU.)

4.26.9 When considering the "EX/MEM" forwarding in 4.26.6, the "EX to 1st" generates no stalls, but "EX to 1st and EX to 2nd" generates one stall. However, "MEM to 1st" and "MEM to 1st and MEM to 2nd" will always generate the same number of stalls. (All "MEM to 1st" dependencies cause a stall, regardless of the type of forwarding. This stall causes the 2nd instruction's ID phase to overlap with the base instruction's WB phase, in which case no forwarding is needed.)

4.27

4.27.1

```
add   x15, x12, x11
nop
nop
ld    x13, 4(x15)
ld    x12, 0(x2)
nop
or    x13, x15, x13
nop
nop
sd    x13, 0(x15)
```

4.27.2 It is not possible to reduce the number of NOPs.

4.27.3 The code executes correctly. We need hazard detection only to insert a stall when the instruction following a load uses the result of the load. That does not happen in this case.

4.27.4

Cycle	1	2	3	4	5	6	7	8	
`add`	IF	ID	EX	ME	WB				
`ld`		IF	ID	EX	ME	WB			
`ld`			IF	ID	EX	ME	WB		
`or`				IF	ID	EX	ME	WB	
`sd`					IF	ID	EX	ME	WB

Because there are no stalls in this code, PCWrite and IF/IDWrite are always 1 and the mux before ID/EX is always set to pass the control values through.

(1) ForwardA = X; ForwardB = X (no instruction in EX stage yet)

(2) ForwardA = X; ForwardB = X (no instruction in EX stage yet)

(3) ForwardA = 0; ForwardB = 0 (no forwarding; values taken from registers)

(4) ForwardA = 2; ForwardB = 0 (base register taken from result of previous instruction)

(5) ForwardA = 1; ForwardB = 1 (base reguster taken from result of two instructions previous)

(6) ForwardA = 0; ForwardB = 2 (rs1 = x15 taken from register; rs2 = x13 taken from result of 1st ld—two instructions ago)

(7) ForwardA = 0; ForwardB = 2 (base register taken from register file. Data to be written taken from previous instruction)

4.27.5 The hazard detection unit additionally needs the values of rd that comes out of the MEM/WB register. The instruction that is currently in the ID stage needs to be stalled if it depends on a value produced by (or forwarded from) the instruction in the EX or the instruction in the MEM stage. So we need to check the destination register of these two instructions. The Hazard unit already has the value of `rd` from the EX/MEM register as inputs, so we need only add the value from the MEM/WB register.

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

The value of `rd` from EX/MEM is needed to detect the data hazard between the `add` and the following `ld`. The value of `rd` form MEM/WB is needed to detect the data hazard between the first `ld` instruction and the `or` instruction.

4.27.6

Cycle	1	2	3	4	5	6
add	IF	ID	EX	ME	WB	
ld		IF	ID	–	–	EX
ld			IF	–	–	ID

(1) PCWrite = 1; IF/IDWrite = 1; control mux = 0

(2) PCWrite = 1; IF/IDWrite = 1; control mux = 0

(3) PCWrite = 1; IF/IDWrite = 1; control mux = 0

(4) PCWrite = 0; IF/IDWrite = 0; control mux = 1

(5) PCWrite = 0; IF/IDWrite = 0; control mux = 1

4.28

4.28.1 The CPI increases from 1 to 1.4125.

An incorrectly predicted branch will cause three instructions to be flushed: the instructions currently in the IF, ID, and EX stages. (At this point, the branch instruction reaches the MEM stage and updates the PC with the correct next instruction.) In other words, 55% of the branches will result in the flushing of three instructions, giving us a CPI of 1 + (1 − 0.45)(0.25)3 = 1.4125. (Just to be clear: the always-taken predictor is correct 45% of the time, which means, of course, that it is incorrect 1 − 0.45 = 55% of the time.)

4.28.2 The CPI increases from 1 to 1.3375. (1 + (.25)(1 − .55) = 1.1125)

4.28.3 The CPI increases from 1 to 1.1125. (1 + (.25)(1 − .85) = 1.0375)

4.28.4 The speedup is approximately 1.019.

Changing half of the branch instructions to an ALU instruction reduces the percentage of instructions that are branches from 25% to 12.5%. Because predicted and mispredicted branches are replaced equally, the misprediction rate remains 15%. Thus, the new CPU is 1 + (.125)(1 − .85) = 1.01875. This represents a speedup of 1.0375 / 1.01875 = 1.0184

4.28.5 The "speedup" is .91.

There are two ways to look at this problem. One way is to look at the two ADD instruction as a branch with an "extra" cycle. Thus, half of the branches have 1 extra cycle; 15% of the other half have 1 extra cycles (the pipeline flush); and the remaining branches (those correctly predicted) have no extra cycles. This gives us a CPI of 1 + (.5)(.25)*1 + (.5)(.25)(.15)*1 = 1.14375 and a speedup of 1.0375 / 1.14375 = .91.

We can also treat the ADD instructions as separate instructions. The modified program now has 1.125n instructions (half of 25% produce

an extra instruction). .125n of these 1.125n instructions (or 11.1%) are branches. The CPI for this new program is 1 + (.111)(.15)*1 = 1.01665. When we factor in the 12.5% increase in instructions, we get a speedup of 1.0375 / (1.125 * 1.01665) = .91.

4.28.6 The predictor is 25% accurate on the remaining branches. We know that 80% of branches are always predicted correctly and the overall accuracy is 0.85. Thus, 0.8*1 + 0.2*x = 0.85. Solving for x shows that x = 0.25.

4.29

4.29.1

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

4.29.2

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

4.29.3 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 of Incorrect (in steady state)	Accuracy
T, NT, T, T, NT	1st occurrence: 0,1,0,1,2 2nd occurrence: 1,2,1,2,3 3rd occurrence: 2,3,2,3,3 4th occurrence: 2,3,2,3,3	C,I,C,C,I	60%

4.29.4 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.

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

4.29.6 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.

4.30

4.30.1

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

4.30.2 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.

4.30.3 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.

4.30.4 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 puts the address in EX, loads the handler's address in MEM, and sets the PC in WB.

4.30.5 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.

4.31.1

```
                     1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
li x12, 0           IF ID EX ME WB
jal ENT             IF ID .. EX ME WB

bne x12, x13, TOP      IF .. ID EX ME WB
slli x5, x12, 3        IF .. ID .. EX ME WB

add x6, x10, x5           IF .. ID EX ME WB
ld x7, 0(x6)              IF .. ID .. EX ME WB

ld x29, 8(x6)                IF .. ID EX ME WB
sub x30, x7, x29             IF .. ID .. .. EX ME WB

add x31, x11, x5                IF .. .. ID EX ME WB
sd x30, 0(x31)                  IF .. .. ID .. EX ME WB

addi x12, x12, 2                           IF .. ID EX ME WB
bne x12, x13, TOP                          IF .. ID .. EX ME WB

slli x5, x12, 3                                  IF .. ID EX ME WB
add x6, x10, x5                                  IF .. ID .. EX ME WB

ld x7, 0(x6)                                           IF .. ID EX ME WB
ld x29, 8(x6)                                          IF .. ID .. EX ME WB

sub x30, x7, x29                                             IF .. ID .. EX ME WB
add x31, x11, x5                                             IF .. ID .. .. EX ME WB

sd x30, 0(x31)                                                     IF .. .. ID EX ME WB
addi x12, x12, 2                                                   IF .. .. ID .. EX ME WB

bne x12, x13, TOP                                                            IF .. ID EX ME WB
slli x5, x12, 3                                                              IF .. ID .. EX ME WB
```

4.31.2 The original code requires 10 cycles/loop on a 1-issue machine (stalls shown below) and 10 cycles/loop on the 2-issue machine. This gives no net speedup. (That's a terrible result considering we nearly doubled the amount of hardware.) We know that the code takes 10 cycles/iteration on the 2-issue machine because the first instruction in loop 1 (The slli) begins execution in cycle 6 and the first instruction in iteration 3 begins execution in cycle 26, so (26–6)/2 = 10.

```
    li x12,0
    jal ENT
TOP:
    slli x5, x12, 3
    add x6, x10, x5
    ld x7, 0(x6)
    ld x29, 8(x6)
    <stall>
    sub x30, x7, x29
    add x31, x11, x5
    sd x30, 0(x31)
    addi x12, x12, 2
ENT:
    bne x12, x13, TOP
```

4.31.3 Here is one possibility:

```
    beqz x13, DONE
    li x12, 0
    jal ENT
TOP:
    slli x5, x12, 3
    add x6, x10, x5
    ld x7, 0(x6)
    ld x29, 8(x6)
    addi x12, x12, 2
    sub x30, x7, x29
    add x31, x11, x5
    sd x30, 0(x31)
ENT:
    bne x12, x13, TOP
DONE:
```

If we switch to a "pointer-based" approach, we can save one cycle/loop.

The code below does this:

```
for (i = 0; i ! = j; i+ = 2) {
  *b = *a - *(a+1);
  b+=2;
  a+=2;
}
```

```
  bez x13, DONE
  li x12, 0
  jal ENT
TOP:
  ld x7, 0(x10)
  ld x29, 8(x10)
  addi x12, x12, 2
  sub x30, x7, x29
  sd x30, 0(x11)
  addi x10, x10, 16
  addi x11, x11, 16
ENT:
  bne x12,x13,TOP
DONE:
```

4.31.4 Here is one possibility:

```
  beqz x13, DONE
  li x12, 0
TOP:
  slli x5, x12, 3
  add x6, x10, x5
  ld x7, 0(x6)
  add x31, x11, x5
  ld x29, 8(x6)
  addi x12, x12, 2
  sub x30, x7, x29
  sd x30, 0(x31)
  bne x12, x13, TOP
DONE:
```

If we switch to a “pointer-based” approach, we can save one cycle/loop.

The code below does this:

```
for (i = 0; i ! = j; i+ = 2) {
   *b = *a - *(a+1);
   b+=2;
   a+=2;
}
   beqz x13, DONE
   li x12, 0
TOP:
   ld x7, 0(x6)
   addi x12, x12, 2
   ld x29, 8(x6)
   addi x6, x6, 16
   sub x30, x7, x29
   sd x30, 0(x31)
   bne x12, x13, TOP
DONE:
```

4.31.5

```
                    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
beqz x13, DONE      IF ID EX ME WB
li x12, 0           IF ID .. EX ME WB

slli x5, x12, 3        IF .. ID EX ME WB
add x6, x10, x5        IF .. ID .. EX ME WB

ld x7, 0(x6)                 IF .. ID EX ME WB
add x31, x11, x5             IF .. ID EX ME WB

ld x29, 8(x6)                      IF ID EX ME WB
addi x12, x12, 2                   IF ID EX ME WB

sub x30, x7, x29                      IF ID .. EX ME WB
sd x30, 0(x31)                        IF ID .. .. EX ME WB

bne x12, x13, TOP                        IF .. .. ID EX ME WB
slli x5, x12, 3                          IF .. .. ID .. EX ME WB

add x6, x10, x5                                  IF .. ID EX ME WB
ld x7, 0(x6)                                     IF .. ID .. EX ME WB

add x31, x11, x5                                       IF .. ID EX ME WB
ld x29, 8(x6)                                          IF .. ID EX ME WB

addi x12, x12, 2                                             IF ID EX ME WB
sub x30, x7, x29                                             IF ID .. EX ME WB

sd x30, 0(x31)                                                  IF .. ID EX ME WB
bne x12, x13, TOP                                               IF .. ID EX ME WB

slli x5, x12, 3                                                       IF ID EX ME WB
add x6, x10, x5                                                       IF ID .. EX ME WB
```

4.31.6 The code from 4.31.3 requires 9 cycles per iteration. The code from 4.31.5 requires 7.5 cycles per iteration. Thus, the speedup is 1.2.

4.31.7 Here is one possibility:

```
      beqz x13, DONE
      li x12, 0
TOP:
      slli x5, x12, 3
      add x6, x10, x5
      add x31, x11, x5
      ld x7, 0(x6)
      ld x29, 8(x6)
      ld x5, 16(x6)
      ld x15, 24(x6)
      addi x12, x12, 4
      sub x30, x7, x29
      sub x14, x5, x15
      sd x30, 0(x31)
      sd x14, 16(x31)
      bne x12, x13, TOP
DONE:
```

4.31.8 Here is one possibility:

```
      beqz x13, DONE
      li x12, 0
      addi x6, x10, 0
TOP:
      ld x7, 0(x6)
      add x31, x11, x5
      ld x29, 8(x6)
      addi x12, x12, 4
      ld x16, 16(x6)
      slli x5, x12, 3
      ld x15, 24(x6)
      sub x30, x7, x29
      sd x30, 0(x31)
      sub x14, x16, x15
      sd x14, 16(x31)
      add x6, x10, x5
      bne x12,x13,TOP
DONE:
```

4.31.9 The code from 4.31.7 requires 13 cycles per unrolled iteration. This is equivalent to 6.5 cycles per original iteration. The code from 4.30.4 requires 7.5 cycles per unrolled iteration. This is equivalent to 3.75 cycles per original iteration. Thus, the speedup is 1.73.

```
                          1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
beqz x13, DONE           IF ID EX ME WB
li x12, 0                IF ID .. EX ME WB

addi x6, x10, 0             IF .. ID EX ME WB
ld x7, 0(x6)                IF .. ID .. EX ME WB

add x31, x11, x5                  IF .. ID EX ME WB
ld x29, 8(x6)                     IF .. ID EX ME WB

addi x12, x12, 4                        IF ID EX ME WB
ld x16, 16(x6)                          IF ID EX ME WB

slli x5, x12, 3                            IF ID EX ME WB
ld x15, 24(x6)                             IF ID EX ME WB

sub x30, x7, x29                              IF ID EX ME WB
sd x30, 0(x31)                                IF ID .. EX ME WB

sub x14, x16, x15                                IF .. ID EX ME WB
sd x14, 16(x31)                                  IF .. ID EX ME WB

add x6, x10, x5                                        IF ID EX ME WB
bne x12,x13,TOP                                        IF ID EX ME WB

ld x7, 0(x6)                                              IF ID EX ME WB
add x31, x11, x5                                          IF ID EX ME WB

ld x29, 8(x6)                                                IF ID EX ME WB
addi x12, x12, 4                                             IF ID EX ME WB

ld x16, 16(x6)                                                  IF ID EX ME WB
slli x5, x12, 3                                                 IF ID EX ME WB

ld x15, 24(x6)                                                     IF ID EX ME WB
sub x30, x7, x29                                                   IF ID EX ME WB

sd x30, 0(x31)                                                        IF ID EX ME WB
sub x14, x16, x15                                                     IF ID .. EX ME WB

sd x14, 16(x31)                                                          IF .. ID EX ME WB
add x6, x10, x5                                                          IF .. ID EX ME WB

bne x12,x13,TOP                                                                IF ID EX ME WB
ld x7, 0(x6)                                                                   IF ID EX ME WB
```

4.31.10 Using the same code as in 4.31.8, the new data path provides no net improvement, because there are no stalls due to structural hazards.

4.32

4.32.1 The energy for the two designs is the same: I-Mem is read, two registers are read, and a register is written. We have: 140pJ + 2*70pJ + 60j = 340pJ.

4.32.2 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; all other instructions result in at most a single 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: 140pJ + 2*70pJ + 60pJ + 140pJ = 480pJ.

4.32.3 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 load instruction results in only one register read (we still must read the register used to generate the address), so our change saves 70pJ (one register read) per load. This is a savings of 70/480 = 14.6%.

4.32.4 `jal` will benefit, because it need not read any registers at all. I-type instructions will also benefit because they need only read one register. If we add logic to detect x0 as a source register, then instructions such as `beqz` (i.e. `beq x0`, …) and `li` (`addi xn, x0`, …) could benefit as well.

4.32.5 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. However, the sum of the latencies for the register read (90ps) and control unit (150ps) are less than the current 250ps cycle time.

4.32.6 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 (for a non-load instruction that writes to a register), or it does not get written to any register (all other instructions, including branches and 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 can affect overall performance if the unused memory reads cause cache misses.

The change also affects energy: A memory read occurs in every cycle instead of only in cycles when a load instruction is in the MEM stage. This increases the energy consumption by 140pJ during 75% of the 250ps clock cycles. This corresponds to a consumption of approximately 0.46 Watts (not counting any energy consumed as a result of cache misses).

4.33

4.33.1 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 the least significant bit of the write register line is stuck at zero, an instruction that writes to an odd-numbered register will end up writing to the even-numbered register. To test for this (1) place a value of 10 in `x1`, 35 in `x2`, and 45 in `x3`, then (2) execute `add x3, x1, x1`. The value of `x3` is supposed to be 20. If bit 0 of the Write Register input to the registers unit is stuck at zero, the value is written to `x2` instead, which means that `x2` will be 40 and `x3` will remain at 45.

4.33.2 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: (1) Place a value of 10 in `x1`, 35 in `x2`, and 45 in `x3`, then (2) execute `add x2, x1, x1`. The value of `x2` is supposed to be 20. If bit 0 of the Write Register input to the registers unit is stuck at 1, the value is written to `x3` instead, which means that `x3` will be 40 and `x2` will remain at 35.

4.33.3 The CPU is still usable. The "simplest" solution is to re-write the compiler so that it uses odd-numbered registers only (not that writing compliers is especially simple). We could also write a "translator" that would convert machine code; however, this would be more complicated because the translator would need to detect when two "colliding" registers are used simultaneously and either (1) place one of the values in an unused register, or (2) push that value onto the stack.

4.33.4 To test for this fault, we need an instruction for which MemRead is set to 1, so it has to be ld. The instruction also needs to have branch set to 0, which is the case for ld. Finally, the instruction needs to have a different result MemRead is incorrectly set to 0. For a load, setting MemRead to 0 results in not reading memory. When this happens, 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.

4.33.5 Only R-type instructions set RegRd to 1. Most R-type instructions would fail to detect this error because reads are non-destructive—the erroneous read would simply be ignored. However, suppose we issued this instruction: `add x1, x0, x0`. In this case, if MemRead were incorrectly set to 1, the data memory would attempt to read the value in memory location 0. In many operating systems, address 0 can only be accessed by a process running in protected/kernel mode. If this is the case, then this instruction would cause a segmentation fault in the presence of this error.

5

Solutions

5.1

5.1.1 2.

5.1.2 I, J, and B[I][0].

5.1.3 A[I][J].

5.1.4 I, J, and B[I][0].

5.1.5 A(J, I) and B[I][0].

5.1.6 32,004 with Matlab. 32,008 with C.

The code references 8*8000 = 64,000 integers from matrix A. At two integers per 16-byte block, we need 32,000 blocks.

The code also references the first element in each of eight rows of Matrix B. Matlab stores matrix data in column-major order; therefore, all eight integers are contiguous and fit in four blocks. C stores matrix data in row-major order; therefore, the first element of each row is in a different block.

5.2

5.2.1

Word Address	Binary Address	Tag	Index	Hit/Miss
0x03	0000 0011	0	3	M
0xb4	1011 0100	b	4	M
0x2b	0010 1011	2	b	M
0x02	0000 0010	0	2	M
0xbf	1011 1111	b	f	M
0x58	0101 1000	5	8	M
0xbe	1011 1110	b	e	M
0x0e	0000 1110	0	e	M
0xb5	1011 0101	b	5	M
0x2c	0010 1100	2	c	M
0xba	1011 1010	b	a	M
0xfd	1111 1101	f	d	M

5.2.2

Word Address	Binary Address	Tag	Index	Offset	Hit/Miss
0x03	0000 0011	0	1	1	M
0xb4	1011 0100	b	2	0	M
0x2b	0010 1011	2	5	1	M
0x02	0000 0010	0	1	0	H
0xbf	1011 1111	b	7	1	M
0x58	0101 1000	5	4	0	M
0xbe	1011 1110	b	7	0	H
0x0e	0000 1110	0	7	0	M
0xb5	1011 0101	b	2	1	H
0x2c	0010 1100	2	6	0	M
0xba	1011 1010	b	5	0	M
0xfd	1111 1101	f	6	1	M

5.2.3

Word Address	Binary Address	Tag	Cache 1		Cache 2		Cache 3	
			Index	Hit/miss	Index	Hit/miss	Index	Hit/miss
0x03	0000 0011	0x00	3	M	1	M	0	M
0xb4	1011 0100	0x16	4	M	2	M	1	M
0x2b	0010 1011	0x05	3	M	1	M	0	M
0x02	0000 0010	0x00	2	M	1	M	0	M
0xbf	1011 1111	0x17	7	M	3	M	1	M
0x58	0101 1000	0x0b	0	M	0	M	0	M
0xbe	1011 1110	0x17	6	M	3	H	1	H
0x0e	0000 1110	0x01	6	M	3	M	1	M
0xb5	1011 0101	0x16	5	M	2	H	1	M
0x2c	0010 1100	0x05	4	M	2	M	1	M
0xba	1011 1010	0x17	2	M	1	M	0	M
0xfd	1111 1101	0x1F	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.

5.3

5.3.1 Total size is 364,544 bits = 45,568 bytes

Each word is 8 bytes; each block contains two words; thus, each block contains 16 = 2^4 bytes.

The cache contains 32KiB = 2^15 bytes of data. Thus, it has 2^15/2^4 = 2^11 lines of data.

Each 64-bit address is divided into: (1) a 3-bit word offset, (2) a 1-bit block offset, (3) an 11-bit index (because there are 2^11 lines), and (4) a 49-bit tag (64 − 3 − 1 − 11 = 49).

The cache is composed of: 2^15 * 8 bits of data + 2^11*49 bits of tag + 2^11*1 valid bits = 364,544 bits.

5.3.2 549,376 bits = 68,672 bytes. This is a 51% increase.

Each word is 8 bytes; each block contains 16 words; thus, each block contains 128 = 2^7 bytes.

The cache contains 64KiB = 2^16 bytes of data. Thus, it has 2^16/2^7 = 2^9 lines of data.

Each 64-bit address is divided into: (1) a 3-bit word offset, (2) a 4-bit block offset, (3) a 9-bit index (because there are 2^9 lines), and (4) a 48-bit tag (64 − 3 − 4 − 9 = 48).

The cache is composed of: 2^16 * 8 bits of data + 2^9*48 bits of tag + 2^9*1 valid bits = 549,376 bits.

5.3.3 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.

5.3.4 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 two-way set associate cache with LRU replacement, even one with a significantly smaller overall capacity, would hit on every access after the first two.

5.4 Yes it is possible. To implement a direct-mapped cache, we need only a function that will take an address as input and produce a 10-bit output. Although it is possible to implement a cache in this manner, it is not clear that such an implementation will be beneficial. (1) The cache would require a larger tag and (2) there would likely be more conflict misses.

5.5

5.5.1 Each cache block consists of four 8-byte words. The total offset is 5 bits. Three of those 5 bits is the word offset (the offset into an 8-byte word). The remaining two bits are the block offset. Two bits allows us to enumerate 2^2 = 4 words.

5.5.2 There are five index bits. This tells us there are 2^5 = 32 lines in the cache.

5.5.3 The ratio is 1.21. The cache stores a total of 32 lines * 4 words/block * 8 bytes/word = 1024 bytes = 8192 bits.

In addition to the data, each line contains 54 tag bits and 1 valid bit. Thus, the total bits required = 8192 + 54*32 + 1*32 = 9952 bits.

5.5.4

Byte Address	Binary Address	Tag	Index	Offset	Hit/Miss	Bytes Replaced
0x00	0000 0000 0000	0x0	0x00	0x00	M	
0x04	0000 0000 0100	0x0	0x00	0x04	H	
0x10	0000 0001 0000	0x0	0x00	0x10	H	
0x84	0000 1000 0100	0x0	0x04	0x04	M	
0xe8	0000 1110 1000	0x0	0x07	0x08	M	
0xa0	0000 1010 0000	0x0	0x05	0x00	M	
0x400	0100 0000 0000	0x1	0x00	0x00	M	0x00-0x1F
0x1e	0000 0001 1110	0x0	0x00	0x1e	M	0x400-0x41F
0x8c	0000 1000 1100	0x0	0x04	0x0c	H	
0xc1c	1100 0001 1100	0x3	0x00	0x1c	M	0x00-0x1F
0xb4	0000 1011 0100	0x0	0x05	0x14	H	
0x884	1000 1000 0100	0x2	0x04	0x04	M	0x80-0x9f

5.5.5 4/12 = 33%.

5.5.6 `<index, tag, data>`

`<0, 3, Mem[0xC00]-Mem[0xC1F]>`

`<4, 2, Mem[0x880]-Mem[0x89f]>`

`<5, 0, Mem[0x0A0]-Mem[0x0Bf]>`

`<7, 0, Mem[0x0e0]-Mem[0x0ff]>`

5.6

[AU1]

5.6.1 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 caches 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.

5.6.2 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.

5.6.3 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.

5.7

5.7.1 When the CPI is 2, there are, on average, 0.5 instruction accesses per cycle. 0.3% of these instruction accesses cause a cache miss (and subsequent memory request). Assuming each miss requests one block, instruction accesses generate an average of 0.5*.003*64 = 0.096 bytes/cycle of read traffic.

Twenty-five percent of instructions generate a read request. Two percent of these generate a cache miss; thus, read misses generate an average of 0.5*0.25*0.02*64 = 0.16 bytes/cycle of read traffic.

Ten percent of instructions generate a write request. Two percent of these generate a cache miss. Because the cache is a write-through cache, only one word (8 bytes) must be written back to memory; but, every write is written through to memory (not just the cache misses). Thus, write misses generate an average of 0.5*0.1*8 = 0.4 bytes/cycle of write traffic. Because the cache is a write-allocate cache, a write miss also makes a read request to RAM. Thus, write misses require an average of 0.5*0.1*0.02*64 = 0.064 bytes/cycle of read traffic.

Hence: The total read bandwidth = 0.096 + 0.16 + 0.064 = 0.32 bytes/cycle, and the total write bandwidth is 0.4 bytes/cycle.

5.7.2 The instruction and data read bandwidth requirement is the same as in 5.7.1.

With a write-back cache, data are only written to memory on a cache miss. But, it is written on every cache miss (both read and write), because any line could have dirty data when evicted, even if the eviction is caused by a read request. Thus, the data write bandwidth requirement becomes 0.5*(0.25 + 0.1)*0.02*0.3*64 = 0.0672 bytes/cycle.

5.8

5.8.1 The addresses are given as word addresses; each 32-bit block contains four words. Thus, every fourth access will be a miss (i.e., a miss rate of 1/4). 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.

5.8.2 The miss rates are 1/2, 1/8, and 1/16, respectively. The workload is exploiting spatial locality.

5.8.3 In this case, the miss rate is 0: The pre-fetch buffer always has the next request ready.

5.9

5.9.1 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.

5.9.2 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.

5.9.3 B = 128 is optimal: Minimizing the miss rate minimizes the total miss latency.

5.10

5.10.1

P1	1.515 GHz
P2	1.11 GHz

5.10.2

P1	6.31 ns	9.56 cycles
P2	5.11 ns	5.68 cycles

For P1, all memory accesses require at least one cycle (to access L1). Eight percent of memory accesses additionally require a 70 ns access to main memory. This is 70/0.66 = 106.06 cycles. However, we can't divide cycles; therefore, we must round up to 107 cycles. Thus, the Average Memory Access time is 1 + 0.08*107 = 9.56 cycles, or 6.31 ps.

For P2, a main memory access takes 70 ns. This is 70/0.90 = 77.78 cycles. Because we can't divide cycles, we must round up to 78 cycles. Thus the Average Memory Access time is 1 + 0.06*78 = 5.68 cycles, or 6.11 ps.

5.10.3

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

For P1, every instruction requires at least one cycle. In addition, 8% of all instructions miss in the instruction cache and incur a 107-cycle delay. Furthermore, 36% of the instructions are data accesses. Eight percent of these 36% are cache misses, which adds an additional 107 cycles.

1 + .08*107 + .36*.08*107 = 12.64

With a clock cycle of 0.66 ps, each instruction requires 8.34 ns.

Using the same logic, we can see that P2 has a CPI of 7.36 and an average of only 6.63 ns/instruction.

5.10.4

AMAT = 9.85 cycles	Worse

An L2 access requires nine cycles (5.62/0.66 rounded up to the next integer).

All memory accesses require at least one cycle. 8% of memory accesses miss in the L1 cache and make an L2 access, which takes nine cycles. 95% of all L2 access are misses and require a 107 cycle memory lookup.

$1 + .08[9 + 0.95*107] = 9.85$

5.10.5

13.04

Notice that we can compute the answer to 5.10.3 as follows: AMAT + %memory * (AMAT-1).

Using this formula, we see that the CPI for P1 with an L2 cache is 9.85 * 0.36*8.85 = 13.04.

5.10.6 Because the clock cycle time and percentage of memory instructions is the same for both versions of P1, it is sufficient to focus on AMAT. We want

AMAT with L2 < AMAT with L1 only

$1 + 0.08[9 + m*107] < 9.56$

This happens when m< .916.

5.10.7 We want P1's average time per instruction to be less than 6.63 ns. This means that we want

(CPI_P1 * 0.66) < 6.63. Thus, we need CPI_P1 < 10.05

CPI_P1 = AMAT_P1 + 0.36(AMAT_P1 – 1).

Thus, we want

AMAT_P1+ 0.36(AMAT_P1-1) < 10.05

This happens when AMAT_P1< 7.65.

Finally, we solve for

$1+ 0.08[9+ m*107] < 7.65$

and find that

m< 0.693

This miss rate can be at most 69.3%.

5.11

5.11.1 Each line in the cache will have a total of six words (two in each of three ways). There will be a total of 48/6 = 8 lines.

5.11.2 T(x) is the tag at index x.

Word Address	Binary Address	Tag	Index	Offset	Hit/Miss	Way 0	Way 1	Way 2
0x03	0000 0011	0x0	1	1	M	T(1)=0		
0xb4	1011 0100	0xb	2	0	M	T(1)=0 T(2)=b		
0x2b	0010 1011	0x2	5	1	M	T(1)=0 T(2)=b T(5)=2		
0x02	0000 0010	0x0	1	0	H	T(1)=0 T(2)=b T(5)=2		
0xbe	1011 1110	0xb	7	0	M	T(1)=0 T(2)=b T(5)=2 T(7)=b		
0x58	0101 1000	0x5	4	0	M	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5		
0xbf	1011 1111	0xb	7	1	H	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5		
0x0e	0000 1110	0x0	7	0	M	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5	T(7)=0	
0x1f	0001 1111	0x1	7	1	M	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5	T(7)=0	T(7)=1
0xb5	1011 0101	0xb	2	1	H	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5	T(7)=0	T(7)=1
0xbf	1011 1111	0xb	7	1	H	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5	T(7)=0	T(7)=1
0xba	1011 1010	0xb	5	0	M	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5	T(7)=2 T(5)=b	T(7)=1
0x2e	0010 1110	0x2	7	0	M	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5	T(7)=2 T(5)=b	T(7)=1
0xce	1100 1110	0xc	7	0	M	T(1)=0 T(2)=b T(5)=2 T(7)=b T(4)=5	T(7)=2 T(5)=b	T(7)=c

5.11.3 No solution given.

5.11.4 Because this cache is fully associative and has one-word blocks, there is no index and no offset. Consequently, the word address is equivalent to the tag.

Word Address	Binary Address	Tag	Hit/Miss	Contents
0x03	0000 0011	0x03	M	3
0xb4	1011 0100	0xb4	M	3, b4
0x2b	0010 1011	0x2b	M	3, b4, 2b
0x02	0000 0010	0x02	M	3, b4, 2b, 2
0xbe	1011 1110	0xbe	M	3, b4, 2b, 2, be
0x58	0101 1000	0x58	M	3, b4, 2b, 2, be, 58
0xbf	1011 1111	0xbf	M	3, b4, 2b, 2, be, 58, bf
0x0e	0000 1110	0x0e	M	3, b4, 2b, 2, be, 58, bf, e
0x1f	0001 1111	0x1f	M	b4, 2b, 2, be, 58, bf, e, 1f
0xb5	1011 0101	0xb5	M	2b, 2, be, 58, bf, e, 1f, b5
0xbf	1011 1111	0xbf	H	2b, 2, be, 58, e, 1f, b5, bf
0xba	1011 1010	0xba	M	2, be, 58, e, 1f, b5, bf, ba
0x2e	0010 1110	0x2e	M	be, 58, e, 1f, b5, bf, ba, 2e
0xce	1100 1110	0xce	M	58, e, 1f, b5, bf, ba, 2e, ce

5.11.5 No solution given.

5.11.6 Because this cache is fully associative, there is no index. (Contents shown in the order the data were accessed. Order does not imply physical location.)

Word Address	Binary Address	Tag	Offset	Hit/Miss	Contents
0x03	0000 0011	0x01	1	M	[2,3]
0xb4	1011 0100	0x5a	0	M	[2,3], [b4,b5]
0x2b	0010 1011	0x15	1	M	[2,3], [b4,b5], [2a,2b]
0x02	0000 0010	0x01	0	H	[b4,b5], [2a,2b], [2,3]
0xbe	1011 1110	0x5f	0	M	[b4,b5], [2a,2b], [2,3], [be, bf]
0x58	0101 1000	0x2c	0	M	[2a,2b], [2,3], [be, bf], [58, 59]
0xbf	1011 1111	0x5f	1	H	[2a,2b], [2,3], [58, 59], [be, bf]
0x0e	0000 1110	0x07	0	M	[2,3], [58, 59], [be, bf], [e,f]
0x1f	0001 1111	0x0f	1	M	[58, 59], [be, bf], [e,f], [1e,1f]
0xb5	1011 0101	0x5a	1	M	[be, bf], [e,f], [1e,1f], [b4, b5]
0xbf	1011 1111	0x5f	1	H	[e,f], [1e,1f], [b4, b5], [be, bf]
0xba	1011 1010	0x5d	0	M	[1e,1f], [b4, b5], [be, bf], [ba, bb]
0x2e	0010 1110	0x17	0	M	[b4, b5], [be, bf], [ba, bb], [2e, 2f]
0xce	1100 1110	0x67	0	M	[be, bf], [ba, bb], [2e, 2f], [ce,cf]

5.11.7 (Contents shown in the order the data were accessed. Order does not imply physical location.)

Word Address	Binary Address	Tag	Offset	Hit/Miss	Contents
0x03	0000 0011	0x01	1	M	[2,3]
0xb4	1011 0100	0x5a	0	M	[2,3], [b4,b5]
0x2b	0010 1011	0x15	1	M	[2,3], [b4,b5], [2a,2b]
0x02	0000 0010	0x01	0	H	[b4,b5], [2a,2b], [2,3]
0xbe	1011 1110	0x5f	0	M	[b4,b5], [2a,2b], [2,3], [be, bf]
0x58	0101 1000	0x2c	0	M	[b4,b5], [2a,2b], [2,3], [58, 59]
0xbf	1011 1111	0x5f	1	M	[b4,b5], [2a,2b], [2,3], [be, bf]
0x0e	0000 1110	0x07	0	M	[b4,b5], [2a,2b], [2,3], [e, f]
0x1f	0001 1111	0x0f	1	M	[b4,b5], [2a,2b], [2,3], [1e, 1f]
0xb5	1011 0101	0x5a	1	H	[2a,2b], [2,3], [1e, 1f], [b4,b5]
0xbf	1011 1111	0x5f	1	M	[2a,2b], [2,3], [1e, 1f], [be, bf]
0xba	1011 1010	0x5d	0	M	[2a,2b], [2,3], [1e, 1f], [ba, bb]
0x2e	0010 1110	0x17	0	M	[2a,2b], [2,3], [1e, 1f], [2e, 2f]
0xce	1100 1110	0x67	0	M	[2a,2b], [2,3], [1e, 1f],, [ce, cf]

5.11.8 Because this cache is fully associative, there is no index.

Word Address	Binary Address	Tag	Offset	Hit/Miss	Contents
0x03	0000 0011	0x01	1	M	[2,3]
0xb4	1011 0100	0x5a	0	M	[2,3], [b4,b5]
0x2b	0010 1011	0x15	1	M	[2,3], [b4,b5], [2a,2b]
0x02	0000 0010	0x01	0	H	[2,3], [b4,b5], [2a,2b]
0xbe	1011 1110	0x5f	0	M	[2,3], [b4,b5], [2a,2b], [be, bf]
0x58	0101 1000	0x2c	0	M	[58,59], [b4,b5], [2a,2b], [be, bf]
0xbf	1011 1111	0x5f	1	H	[58,59], [b4,b5], [2a,2b], [be, bf]
0x0e	0000 1110	0x07	0	M	[e,f], [b4,b5], [2a,2b], [be, bf]
0x1f	0001 1111	0x0f	1	M	[1e,1f], [b4,b5], [2a,2b], [be, bf]
0xb5	1011 0101	0x5a	1	H	[1e,1f], [b4,b5], [2a,2b], [be, bf]
0xbf	1011 1111	0x5f	1	H	[1e,1f], [b4,b5], [2a,2b], [be, bf]
0xba	1011 1010	0x5d	0	M	[1e,1f], [b4,b5], [ba,bb], [be, bf]
0x2e	0010 1110	0x17	0	M	[1e,1f], [b4,b5], [2e,2f], [be, bf]
0xce	1100 1110	0x67	0	M	[1e,1f], [b4,b5], [ce,cf], [be, bf]

5.12

5.12.1 Standard memory time: Each cycle on a 2-Ghz machine takes 0.5 ps. Thus, a main memory access requires 100/0.5 = 200 cycles.

- L1 only: 1.5 + 0.07*200 = 15.5
- Direct mapped L2: 1.5 + .07 × (12 + 0.035 × 200) = 2.83
- 8-way set associated L2: 1.5 + .07 × (28 + 0.015 × 200) = 3.67.

Doubled memory access time (thus, a main memory access requires 400 cycles)

- L1 only: 1.5 + 0.07*400 = 29.5 (90% increase)
- Direct mapped L2: 1.5 + .07 × (12 + 0.035 × 400) = 3.32 (17% increase)
- 8-way set associated L2: 1.5 + .07 × (28 + 0.015 × 400) = 3.88 (5% increase)

5.12.2 $1.5 = 0.07 \times (12 + 0.035 \times (50 + 0.13 \times 200)) = 1.03$

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

5.12.3 No size will achieve the performance goal.

We want the CPI of the CPU with an external L2 cache to be at most 2.83. Let x be the necessary miss rate.

1.5 + 0.07*(50 + x*200) < 2.83

Solving for x gives that $x < -0.155$. This means that even if the miss rate of the L2 cache was 0, a 50-ns access time gives a CPI of 1.5 + 0.07*(50 + 0*200) = 5, which is greater than the 2.83 given by the on-chip L2 caches. As such, no size will achieve the performance goal.

5.13

5.13.1

3 years and 1 day	1096 days	26304 hours

5.13.2

1095/1096 = 99.90875912%

5.13.3 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.

5.13.4 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.

5.14

5.14.1 9. For SEC, we need to find minimum p such that $2^p >= p + d + 1$ and then add one. That gives us p = 8. We then need to add one more bit for SEC/DED.

5.14.2 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 better cost/performance ratio.

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

5.15 Instructors can change the disk latency, transfer rate and optimal page size for more variants. Refer to Jim Gray's paper on the 5-minute rule 10 years later.

5.15.1 32 KB.

To solve this problem, I used the following gnuplot command and looked for the maximum:

```
plot [16:128] log((x*1024/128) *0.7)/(log(2)*(10 + 0.1*x))
```

5.15.2 Still 32 KB. (Modify the gnuplot command above by changing 0.7 to 0.5.)

5.15.3 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.

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

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

5.15.4 (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.

5.16

5.16.1

Address	Virtual Page	TLB H/M	TLB		
			Valid	Tag	Physical Page
4669 0x123d	1	TLB miss PT hit PF	1	b	12
			1	7	4
			1	3	6
			1 (last access 0)	1	13
2227 0x08b3	0	TLB miss PT hit	1 (last access 1)	0	5
			1	7	4
			1	3	6
			1 (last access 0)	1	13
13916 0x365c	3	TLB miss PT hit	1 (last access 1)	0	5
			1	7	4
			1 (last access 2)	3	6
			1 (last access 0)	1	13
34587 0x871b	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 0xbee6	b	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)	b	12
12608 0x3140	3	TLB hit PT hit	1 (last access 1)	0	5
			1 (last access 3)	8	14
			1 (last access 5)	3	6
			1 (last access 4)	b	12
49225 0xc040	c	TLB miss PT hit PF	1 (last access 6)	c	15
			1 (last access 3)	8	14
			1 (last access 5)	3	6
			1 (last access 4)	b	12

5.16.2

Address	Virtual Page	TLB H/M	TLB		
			Valid	Tag	Physical Page
4669 0x123d	1	TLB miss PT hit	1	11	12
			1	7	4
			1	3	6
			1 (last access 0)	0	5
2227 0x08b3	0	TLB hit	1	11	12
			1	7	4
			1	3	6
			1 (last access 1)	0	5
13916 0x365c	0	TLB hit PT hit	1	11	12
			1	7	4
			1	3	6
			1 (last access 2)	0	5
34587 0x871b	2	TLB miss PT hit PF	1 (last access 3)	2	13
			1	7	4
			1	3	6
			2	0	5
48870 0xbee6	2	TLB hit PT hit	1 (last access 4)	2	13
			1	7	4
			1	3	6
			1 (last access 2)	0	5
12608 0x3140	0	TLB hit PT hit	1 (last access 4)	2	13
			1	7	4
			1	3	6
			5	0	5
49225 0xc040	3	TLB hit PT hit	1 (last access 4)	2	13
			1	7	4
			1 (last access 6)	3	6
			1 (last access 5)	0	5

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

5.16.3 Two-way set associative

Address	Virtual Page	Tag	Index	TLB H/M	TLB			
					Valid	Tag	Physical Page	Index
4669 0x123d	1	0	1	TLB miss PT hit PF	1	b	12	0
					1	7	4	1
					1	3	6	0
					1 (last access 0)	0	13	1
2227 0x08b3	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 0x365c	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 0x871b	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 0xbee6	b	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 0x3140	3	1	1	TLB hit PT 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 0xc049	c	6	0	TLB miss PT miss PF	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

5.16.4 Direct mapped

Address	Virtual Page	Tag	Index	TLB H/M	TLB			
					Valid	Tag	Physical Page	Index
4669 0x123d	1	0	1	TLB miss PT hit PF	1	b	12	0
					1	0	13	1
					1	3	6	2
					0	4	9	3
2227 0x08b3	0	0	0	TLB miss PT hit	1	0	5	0
					1	0	13	1
					1	3	6	2
					0	4	9	3
13916 0x365c	3	0	3	TLB miss PT hit	1	0	5	0
					1	0	13	1
					1	3	6	2
					1	0	6	3
34587 0x871b	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 0xbee6	b	2	3	TLB miss PT hit	1	2	14	0
					1	0	13	1
					1	3	6	2
					1	2	12	3
12608 0x3140	3	0	3	TLB hit PT hit	1	2	14	0
					1	0	13	1
					1	3	6	2
					1	0	6	3
49225 0xc049	c	3	0	TLB miss PT miss PF	1	3	15	0
					1	0	13	1
					1	3	6	2
					1	0	6	3

5.16.5 Without a TLB, almost every memory access would require two accesses to RAM: An access to the page table, followed by an access to the requested data.

5.17

5.17.1 The tag size is 32–log2(8192) = 32–13 = 19 bits. All five page tables would require 5 × (2^19 × 4) bytes = 10 MB.

5.17.2 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 contains 2^(19 – 8) = 2048 entries, requiring 2048 × 4 = 8 KB each and covering 2048 × 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 1st-level 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.

5.17.3 The page index is 13 bits (address bits 12 down to 0).

A 16 KB direct-mapped cache with two 64-bit words per block would have 16-byte blocks and thus 16 KB/16 bytes = 1024 blocks. Thus, it would have 10 index bits and 4 offset bits and the index would extend outside of the page index.

The designer could increase the cache's associativity. This would reduce the number of index bits so that the cache's index fits completely inside the page index.

5.18

5.18.1 Worst case is 2^(43 − 12) = 2^31 entries, requiring 2^(31) × 4 bytes = 2^33 = 8 GB.

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

5.18.3 Yes, if segment table entries are assumed to be the physical page numbers of segment pages, and one bit is reserved as the valid bit, then each one has an effective reach of (2^{31}) * 4KiB = 8TiB, which is more than enough to cover the physical address space of the machine (16 GiB).

5.18.4 Each page table level contains 4KiB/4B = 1024 entries, and so translates log2(1024) = 10 bits of virtual address. Using 43-bit virtual addresses and 4KiB pages, we need ceil((43 − 12)/10) = 4 levels of translation.

5.18.5 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.

5.19

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

5.19.2 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.

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

5.20

5.20.1 There are no hits.

5.20.2 Direct mapped

0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0
M	M	M	M	M	M	M	M	H	H	M	M	M	M	H	H	M

5.20.3 Answers will vary.

5.20.4 MRU is an optimal policy.

5.20.5 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.

5.20.6 If you knew that an address had limited temporal locality and would conflict with another block in the cache, choosing not to cache it could improve the miss rate. On the other hand, you could worsen the miss rate by choosing poorly which addresses to cache.

5.21

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

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

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

The CPI of a machine running on native hardware is 1.5 + 120/10000*15 = 1.68. To keep the performance degradation to 10%, we need

1.5 + 120/10000*(15 + x) < 1.1*1.68

Solving for x shows that a trap to the VMM can take at most 14 cycles.

5.21.2 Non-virtualized CPI = 1.5 + 120/10000 × 15 + 30/10000 × 1100 = 4.98

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

Non-virtualized CPI with half I/O = 1.5 + 120/10000 × 15 + 15/10000 × 1100 = 3.33

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

5.22 Virtual memory aims to provide each application with the illusion of the entire address space of the machine. Virtual machines aim to provide each operating system with the illusion of having the entire machine at 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.

5.23 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 degradation 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 recent Intel processors that do micro-op fusion, allowing several instructions to be handled by fewer instructions.

5.24

5.24.1 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.

5.24.2 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.

5.24.3 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.

5.25

5.25.1 There are six possible orderings for these instructions.

Ordering 1:

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

Result: (5,5)

Ordering 2:

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

Result: (5,5)

Ordering 3:

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

Result: (6,3)

Ordering 4:

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

Result: (5,3)

Ordering 5:

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

Result: (6,5)

Ordering 6:

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

Result: (6,3)

If coherency isn't ensured:

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

5.25.2 Direct mapped

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 write X block in cache		X block is invalided
X[1] = 3;	write X block in cache		

5.25.3 Best case:

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

Worst case:

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

5.25.4

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)

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

Result: (2,0).

5.25.6 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.

5.26

5.26.1 Benchmark A

AMATprivate = 1 + 0.03*[5 + 0.1*180] = 1.69

AMATshared = 1 + 0.03*[20 + 0.04*180] = 1.82

Benchmark B

AMATprivate = 1+ 0.03*[5 + 0.02*180] = 1.26

AMATshared = 1+ 0.03*[20 + 0.01*180] = 1.65

Private cache is superior for both benchmarks.

5.26.2 In a private cache system, the first link of the chip is the link from the private L2 caches to memory. Thus, the memory latency doubles to 360. In a shared cache system, the first link off the chip is the link to the L2 cache. Thus, in this case, the shared cache latency doubles to 40.

Benchmark A

$AMAT_{private} = 1 + .03*[5 + .1*360] = 2.23$

$AMAT_{shared} = 1 + .03*[40 + .04*180] = 2.416$

Benchmark B

$AMAT_{private} = 1 + .03*[5 + .02*360] = 1.37$

$AMAT_{shared} = 1 + .03*[40 + .01*180] = 2.25$

Private cache is superior for both benchmarks.

5.26.3

	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. No disadvantage.	Threads often have private working sets, and using a private L2 prevents cache contamination and conflict misses between threads.
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.

5.26.4 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.

5.26.5 Four times.

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

5.27

5.27.1 `srcIP` and `refTime` fields. Two misses per entry.

5.27.2 Group the `srcIP` and `refTime` fields into a separate array. (I.e., create two parallel arrays. One with `srcIP` and `refTime`, and the other with the remaining fields.)

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

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

5.28

5.28.1 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.

5.28.2 Answers will vary depending on which data set is used.

5.28.3 Answers will vary.

5.29

5.29.1 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.

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

Native: L; NPT: $L \times (L + 2)$.

5.29.3 Shadow page table: page fault rate.

NPT: TLB miss rate.

5.29.4 Shadow page table: 1.03

NPT: 1.04.

5.29.5 Combining multiple page table updates.

5.29.6 NPT caching (similar to TLB caching).

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) 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.

Finish 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, 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 of 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 As shown below, each iteration of the loop requires 16 cycles. The loop runs 999 times. Thus, the total number of cycles is $16 \times 999 + 3 = 15{,}984$.

```
        li    x5, 8000
        add   x12, x10, x5
        addi  x11, x10, 16
LOOP:   fld   f0, -16(x11)
        fld   f1, -1(x11)
        stall
        stall
        stall
        stall
        stall
        stall
        fadd.d f2, f0, f1
        stall
        stall
        stall
        stall
        fsd   f2, 0(x11)
        addi  x11, x11, 8
        ble   x11, x12, LOOP
```

6.4.2 The following code removes one stall per iteration:

a.
```
        li    x5, 8000
        add   x12, x10, x5
        addi  x11, x10, 16
LOOP:   fld   f0, -16(x11)
        fld   f1, -1(x11)
        stall
        stall
        stall
        stall
        stall
        stall
        fadd.d f2, f0, f1
        addi  x11, x11, 8
        stall
        stall
        stall
        fsd   f2, -8(x11)
        ble   x11, x12, LOOP
```

b. Thus, the new loop takes $15 \times 999 = 14{,}958$ cycles.

6.4.3 Array elements `D[j]` and `D[j-1]` will have loop carried dependencies. The value loaded into `D0` during iteration `i` was produced during iteration `i-1`.

6.4.4

```
        li     x5, 8000
        add    x12, x10, x5
        fld    f0, 0(x11)
        fld    f1, 8(x11)
        addi   x11, x11, 8
        stall
        stall
        stall
        stall
        stall
LOOP:   fadd.d f2, f0, f1
        addi   x11, x11, 8
        fmv.d f0, f1
        fmv.d f1, f2
        stall
        fsd    f2, 0(x11)
        ble    x11, x12, LOOP
```

This loop takes seven cycles and runs 999 times. Thus, the total number of cycles is $7 \times 999 + 10 = 7003$.

6.4.5

```
        fld   f0, 0(x11)
        fld   f1, 8(x11)
        li    x5, 8000
        add   x12, x10, x5
        addi  x11, x11, 16
        stall
        stall
        stall
LOOP:   fadd.d f2, f0, f1
        stall
        stall
        stall
        stall
        fadd.d f0, f2, f1
        fsd   f2, 0(x11)
        stall
        stall
        stall
        fadd.d f1, f2, f0
        fsd   f0, 8(x11)
        addi  x11, x11, 24
        stall
```

```
stall
fsd   f1, -8(x11)
bne   x11, x12, LOOP
```

The unrolled loop takes 17 cycles, but runs only 333 times. Thus, the total number of cycles is $17 \times 333 + 10 = 5671$.

6.4.6 Include two copies of the loop: The unrolled loop and the original loop. Suppose you unrolled the loop U times. Run the unrolled loop until the number of iterations left is less than U. (In some sense your unrolled loop will be doing this: `for (i = 0; i + U < MAX; i+= U)`.) At this point, switch to the unrolled loop. (In some sense, your original loop will be doing this: `for (; i < MAX; i++)`.)

6.4.7 It is not possible to use message passing to improve performance—even if the message passing system has no latency. There is simply not enough work that can be done in parallel to benefit from using multiple CPUs. All the work that can be done in parallel can be scheduled between dependent floating point instructions.

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 four-core system. The computations involved are: $(m \times p \times n)$ multiplications and $(m \times p \times (n-1))$ 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 four 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 Answer is same as 6.9.1.

6.9.3

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

6.9.4

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

6.10 This is an open-ended question.

6.10.1 There is no solution to this problem (it is an open-ended question-no need to change the solutions document).

6.10.2 There is no solution to this problem (it is an open-ended question-no need to change the solutions document).

6.11

6.11.1 The answer should include an RISC-V program that includes four 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 four 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, an 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.

6.12.1 There is no solution to this problem (it is an open-ended question-no need to change the solutions document).

6.12.2 There is no solution to this problem (it is an open-ended question-no need to change the solutions document).

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

6.13.1 There is no solution to this problem (it is an open-ended question-no need to change the solutions document).

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

6.14.1 There is no solution to this problem (it is an open-ended question-no need to change the solutions document).

6.14.2 There is no solution to this problem (it is an open-ended question-no need to change the solutions document).

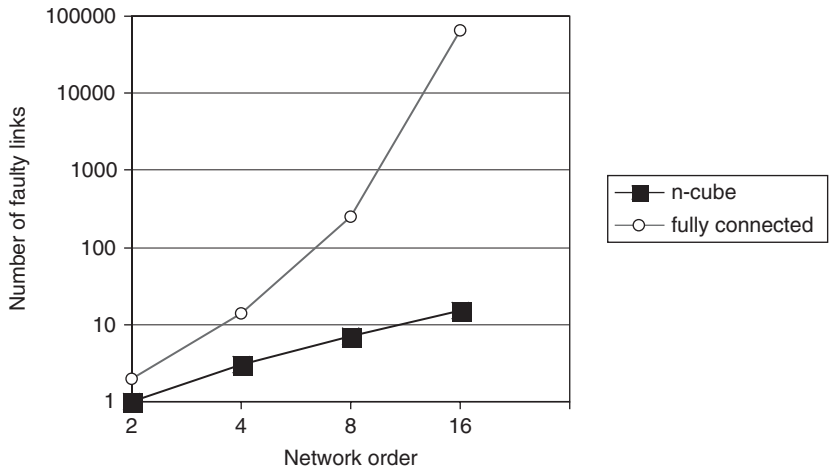
6.15 This question will require the students to research on the Internet both the AMD Fusion architecture and the Intel QuickPath technology. The key is that students become aware of these technologies. The actual bandwidth and latency values should be available right off the company websites, and will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

6.15.1 There is no solution to this problem (it is a lab-based question-no need to change the solutions document).

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 nonzero 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 two loads, eight matrix elements in parallel from A and eight matrix elements from B, into a single vector register and then perform a vector subtract. We would then issue two vector stores to put the result in memory.

Since the vector processor does not have comparison instructions, we would have CPU A perform two 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.

6.20 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 four cores contributing, we would see an average latency of 0.8 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.

Solutions for Appendix A Exercises

A.1

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

A.2 Here is the first equation:

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

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

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

Now distribute the last factor:

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

Now distribute within each term; we show one example:

$$((A \cdot B) \cdot (\overline{A} + \overline{B} + \overline{C})) = (A \cdot B \cdot \overline{A}) + (A \cdot B \cdot \overline{B}) + (A \cdot B \cdot \overline{C}) = 0 + 0 + (A \cdot B \cdot \overline{C})$$

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

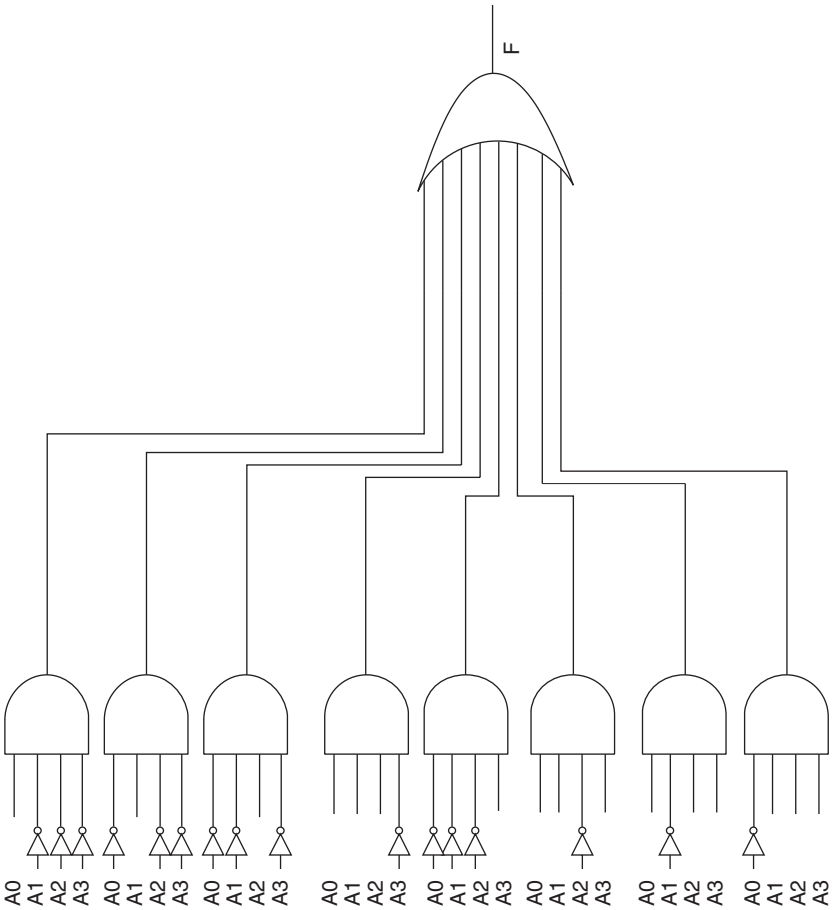
$$E = (A \cdot B \cdot \overline{C}) + (A \cdot \overline{B} \cdot C) + (\overline{A} \cdot B \cdot C),$$ which is the desired result.

A.7 Four inputs A0–A3 & F (O/P) = 1 if an odd number of 1s exist in A.

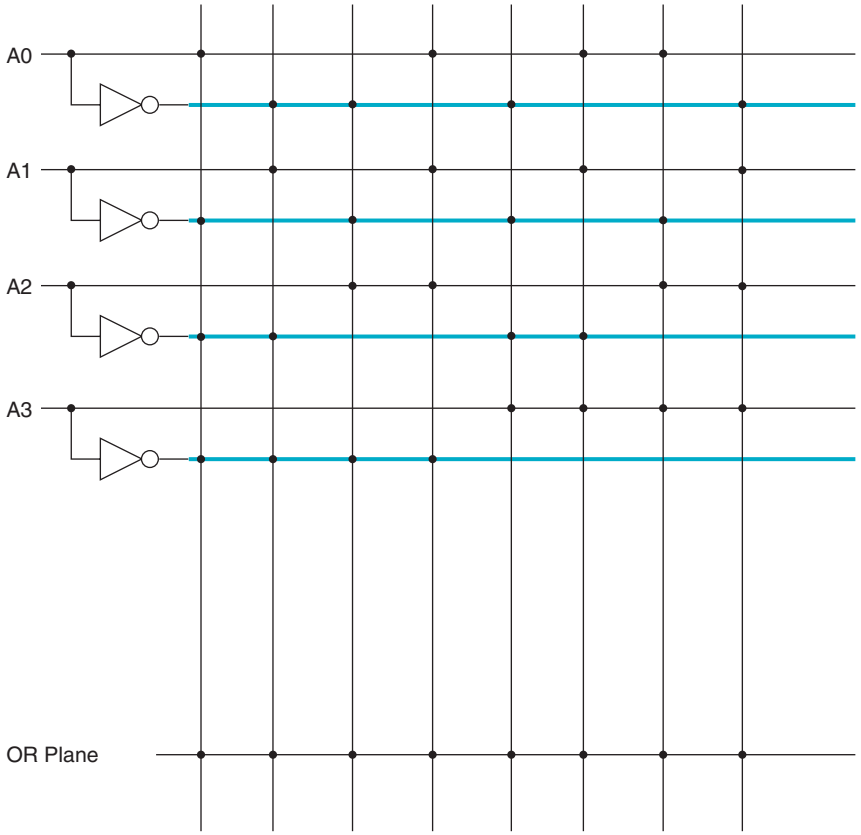
A3	A2	A1	A0	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

A.8 F = A3′A2′A1′A0 + A3′A2′A1 A0′ + A3′A2 A1′A0' + A3′A2 A1 A0 +
A3 A2′A1′A0′ + A3 A2′A1 A0 + A3′A2′A1 A0′ + A3 A2 A1 A0′

Note: F = A0 XOR A1 XOR A2 XOR A3. Another question can ask the students to prove that.


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

A.9



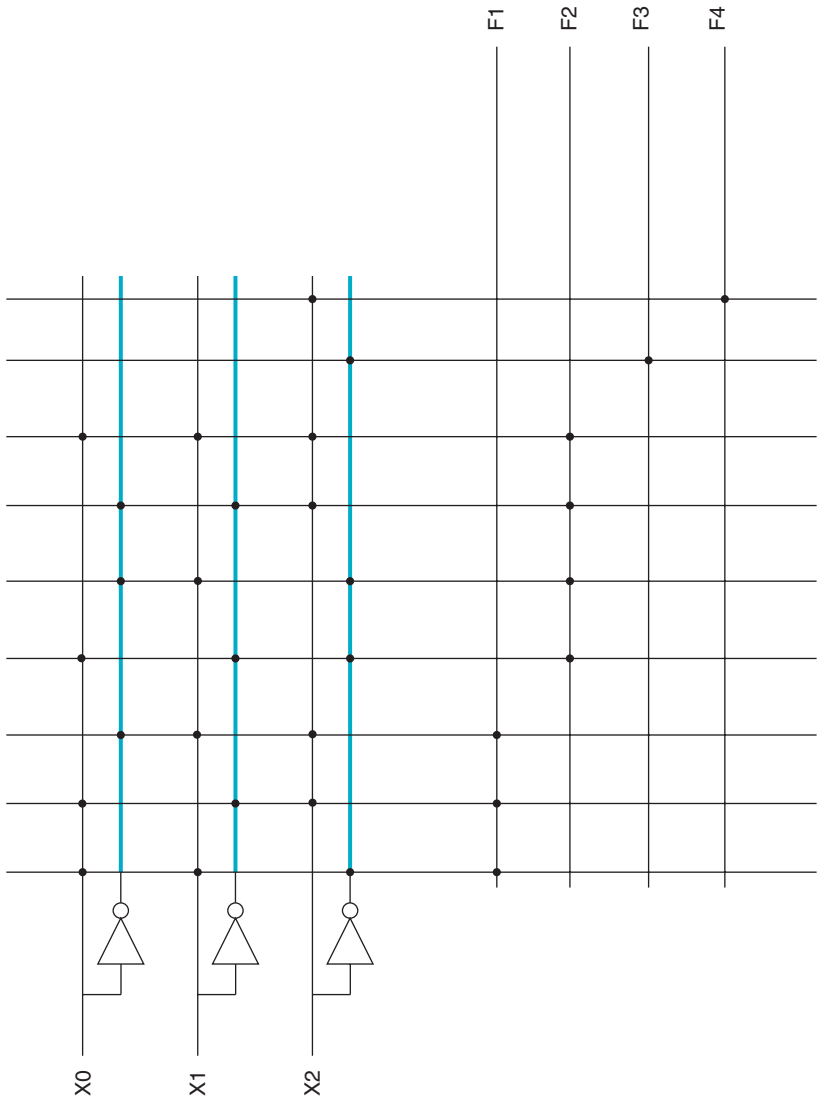
A.10 No solution provided.

A.11

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

F1 = X2′X1 X0 + X2 X1′X0 + X2 X1 X0′
F2 = X2′X1′X0 + X2′X1 X0′ + X2 X1′X0′ + X2 X1 X0 = (A XOR B XOR C)
F3 = X2′
F4 = X2 (= F3′)

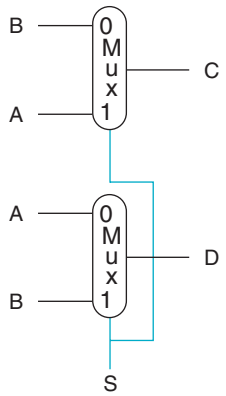
A.12



A.13

- $\overline{x2}y2 + x2y2\overline{x1}y1 + x2y2x1y1\overline{x0}y0 + \overline{x2y2x1}y1 + \overline{x2y2x1y1x0}y0 + \overline{x2y2}x1y1\overline{x0}y0 + x2y2\overline{x1y1x0}y0 + x2y2x1y1\overline{x0}y0$
- $x2\overline{y2} + x2y2\overline{x1}y1 + x2y2x1y1\overline{x0}y0 + \overline{x2y2x1}y1 + x2y2\overline{x1y1x0}y0 + \overline{x2y2x1y1x0}y0 + \overline{x2y2}x1y1\overline{x0}y0$
- $(x2y2 + \overline{x2}\,\overline{y2})(x1y1 + \overline{x1}\,\overline{y1})(x0y0 + \overline{x0}\,\overline{y0})$

A.14



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

Similarly,

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

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

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

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

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

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

A.16 No solution provided.

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

A.19

```
module LATCH(clock,D,Q,Qbar)
input clock,D;
reg Q;
wire Qbar;
assign Qbar = ~Q;
always @(D,clock)  //senstivity list watches clock and data
begin
   if(clock)
     Q = D;
end
endmodule
```

A.20

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

always @ (enable, in)
  if (enable) begin
  out = 0;
  case (in)
         2'h0 : out = 4'h1;
         2'h1 : out = 4'h2;
         2'h2 : out = 4'h4;
         2'h3 : out = 4'h8;
  endcase
end
endmodule
```

A.21

```
module ACC(Clk, Rst, Load, IN, LOAD, OUT);

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

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

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

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

if Load begin
         Register = LOAD;
end
end

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

endmodule
```

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

```
module MULT(clk, load, Multiplicand, Multiplier, Product, done);
input clk, load;
input [31:0] Multiplicand, Multiplier;
output [63:0] Product;
output done;

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

        initial begin
          done = 0; loop = 0;
        end

        always @(posedge clk) begin
          if (load && loop ==0) begin
                  done <= 0;
                  Product <=0;
                  A <= Multiplicand;
                  B <= Multiplier;
                  loop <= 32;
        end

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

            A <= A << 1;
            B <= B >> 1;
            loop <= loop -1;

            if(loop == 0)
              done <= 1;
          end

        end
        endmodule
```

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

```
module DIV(clk, load, Divisor, Dividend, Quotient, Remainder, done);

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

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

initial begin
  done = 0; loop = 0;
end

assign Remainder = R[31:0];

always @(posedge clk) begin
  if (load && loop ==0) begin
    done <= 0;
    R <=Dividend;
    D <= Divisor << 32;
        Quotient <=0;
        loop <= 33;
end

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

        D <= D >> 1;
        loop <= loop - 1;
```

```
            if(loop == 0)
              done <= 1;
        end

    end
    endmodule
```

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

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

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

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

$$LessThan = Overflow \oplus Result31$$

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

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

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

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

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

A.26 C1 = c4, C2 = c8, C3 = c12, and C4 = c16.

$c4 = G_{3,0} + (P_{3,0} \cdot c0)$.

c8 is given in the exercise.

$c12 = G_{11,8} + (P_{11,8} \cdot G_{7,4}) + (P_{11,8} \cdot P_{7,4} \cdot G_{3,0}) + (P_{11,8} \cdot P_{7,4} \cdot P_{3,0} \cdot c0)$.

$c16 = G_{15,12} + (P_{15,12} \cdot G_{11,8}) + (P_{15,12} \cdot P_{11,8} \cdot G_{7,4})$
$+ (P_{15,12} \cdot P_{11,8} \cdot P_{7,4} \cdot G_{3,0}) + (P_{15,12} \cdot P_{11,8} \cdot P_{7,4} \cdot P_{3,0} \cdot c0)$.

A.27 The equations for c4, c8, and c12 are the same as those given in the solution to Exercise 4.44. Using 16-bit adders means using another level of carry lookahead logic to construct the 64-bit adder. The second level generate, G0′, and propagate, P0′, are

$$G0' = G_{15,0} = G_{15,12} + P_{15,12} \cdot G_{11,8} + P_{15,12} \cdot P_{11,8} \cdot G_{7,4} + P_{15,12} \cdot P_{11,8} \cdot P_{7,4} \cdot G_{3,0}$$

and

$$P0' = P_{15,0} = P_{15,12} \cdot P_{11,8} \cdot P_{7,4} \cdot P_{3,0}$$

Using G0′ and P0′, we can write c16 more compactly as

$$c16 = G_{15,0} + P_{15,0} \cdot c0$$

and

$$c32 = G_{31,16} + P_{31,16} \cdot c16$$
$$c48 = G_{47,32} + P_{47,32} \cdot c32$$
$$c64 = G_{63,48} + P_{63,48} \cdot c48$$

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

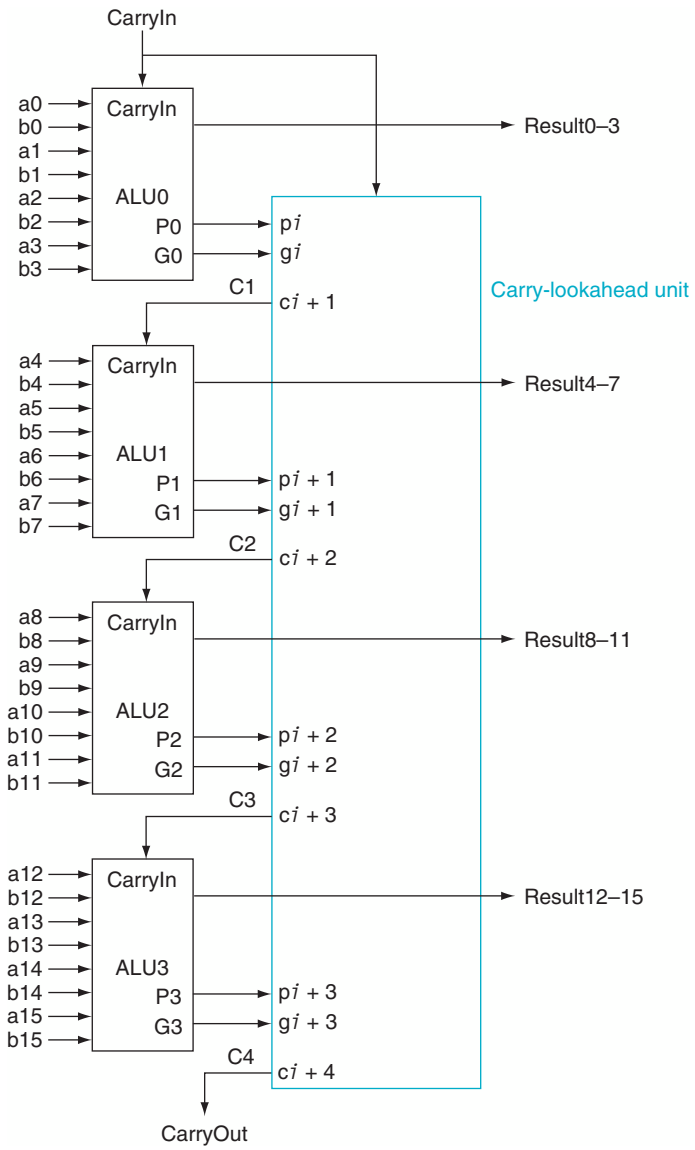


FIGURE A.6.3 **Four 4-bit ALUs using carry lookahead to form a 16-bit adder.** Note that the carries come from the carry-lookahead unit, not from the 4-bit ALUs.

A.28 No solution provided.

A.29 No solution provided.

A.30 No solution provided.

A.31 No solution provided.

A.32 No solution provided.

A.33 No solution provided.

A.34 The longest paths through the top (ripple carry) adder organization in Figure A.14.1 all start at input a0 or b0 and pass through seven full adders on the way to output s4 or s5. There are many such paths, all with a time delay of $7 \times 2T = 14T$. The longest paths through the bottom (carry save) adder all start at input b0, e0, f0, b1, e1, or f1 and proceed through six full adders to outputs s4 or s5. The time delay for this circuit is only $6 \times 2T = 12T$.