1 Solutions

Solution 1.1

1.1.1   Computer used to run large problems and usually accessed via a network:
5 supercomputers

1.1.2 10^{15} or 2^{50} bytes: 7 petabyte

1.1.3   Computer composed of hundreds to thousands of processors and terabytes
of memory: 3 servers

1.1.4   Today's science fiction application that probably will be available in near
future: 1 virtual worlds

1.1.5   A kind of memory called random access memory: 12 RAM

1.1.6   Part of a computer called central processor unit: 13 CPU

1.1.7   Thousands of processors forming a large cluster: 8 datacenters

1.1.8   A microprocessor containing several processors in the same chip: 10 multi-
core processors

1.1.9   Desktop computer without screen or keyboard usually accessed via a net-
work: 4 low-end servers

1.1.10   Currently the largest class of computer that runs one application or one
set of related applications: 9 embedded computers

1.1.11   Special language used to describe hardware components: 11 VHDL

1.1.12   Personal computer delivering good performance to single users at low
cost: 2 desktop computers

1.1.13   Program that translates statements in high-level language to assembly
language: 15 compiler

1.1.14   Program that translates symbolic instructions to binary instructions: 21 assembler

1.1.15   High-level language for business data processing: 25 cobol

1.1.16   Binary language that the processor can understand: 19 machine language

1.1.17   Commands that the processors understand: 17 instruction

1.1.18   High-level language for scienti? c computation: 26 fortran

1.1.19   Symbolic representation of machine instructions: 18 assembly language

1.1.20   Interface between user   ' s program and hardware providing a variety of services and supervision functions: 14 operating system

1.1.21   Software/programs developed by the users: 24 application software

1.1.22   Binary digit (value 0 or 1): 16 bit

1.1.23   Software layer between the application software and the hardware that includes the operating system and the compilers: 23 system software

1.1.24   High-level language used to write application and system software: 20 C

1.1.25   Portable language composed of words and algebraic expressions that must be translated into assembly language before run in a computer: 22 high-level language

1.1.26 10^{12} or 2^{40} bytes: 6 terabyte

Solution 1.2

1.2.1   8 bits $\times$ 3 colors $=$ 24 bits/pixel $=$ 4 bytes/pixel. 1280$\times$ 800 pixels$=$ 1,024,000 pixels. 1,024,000 pixels $\times$ 4 bytes/pixel $=$ 4,096,000 bytes (approx 4 Mbytes).

1.2.2   2 GB $=$ 2000 Mbytes. No. frames $=$ 2000 Mbytes/4 Mbytes $=$ 500 frames.

1.2.3   Network speed: 1 gigabit network $==>$ 1 gigabit/per second $=$ 125 Mbytes/ second. File size: 256 Kbytes$=$ 0.256 Mbytes. Time for 0.256 Mbytes $=$ 0.256/125 $=$ 2.048 ms.

1.2.4   2 microseconds from cache ==> 20 microseconds from DRAM. 20 micro-
seconds from DRAM ==> 2 seconds from magnetic disk. 20 microseconds from
DRAM ==> 2 ms from ? ash memory.

Solution 1.3

1.3.1   P2 has the highest performance

performance of P1 (instructions/sec) $= 2 \times 10^9/1.5 = 1.33 \times 10^9$
performance of P2 (instructions/sec) $= 1.5 \times 10^9/1.0 = 1.5 \times 10^9$
performance of P3 (instructions/sec) $= 3 \times 10^9/2.5 = 1.2 \times 10^9$

1.3.2   No. cycles$=$ time $\times$ clock rate

cycles(P1) $= 10 \times 2 \times 10^9 = 20 \times 10^9$ s
cycles(P2) $= 10 \times 1.5 \times 10^9 = 15 \times 10^9$ s
cycles(P3) $= 10 \times 3 \times 10^9 = 30 \times 10^9$ s

time $=$ (No. instr. $\times$ CPI)/clock rate, then No. instructions $=$ No. cycles/CPI

instructions(P1) $= 20 \times 10^9/1.5 = 13.33 \times 10^9$
instructions(P2) $= 15 \times 10^9/1 = 15 \times 10^9$
instructions(P3) $= 30 \times 10^9/2.5 = 12 \times 10^9$

1.3.3   time$_{new}$ = time$_{old}$ $\times$ 0.7 = 7 s

CPI = CPI $\times$ 1.2, then CPI(P1) $= 1.8$, CPI(P2) $= 1.2$, CPI(P3) $= 3$

? = No. instr. $\times$ CPI/time, then

?(P1) $= 13.33 \times 10^9 \times 1.8/7 = 3.42$ GHz
?(P2) $= 15 \times 10^9 \times 1.2/7 = 2.57$ GHz
?(P3) $= 12 \times 10^9 \times 3/7 = 5.14$ GHz

1.3.4   IPC = 1/CPI = No. instr./(time $\times$ clock rate)

IPC(P1) $= 1.42$
IPC(P2) $= 2$
IPC(P3) $= 3.33$

1.3.5   Time$_{new}$/Time$_{old}$ = 7/10 = 0.7. So?$_{new}$ = ?$_{old}$/0.7 = 1.5 GHz/0.7 = 2.14 GHz.

1.3.6   Time$_{new}$/Time$_{old}$ = 9/10 = 0.9.
So Instructions$_{new}$ = Instructions$_{old}$ $\times$ 0.9 $= 30 \times 10^9 \times 0.9 = 27 \times 10^9$.

Solution 1.4

1.4.1   P2

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. $\times$ CPI/clock rate

P1: Time class A = 0.66×10^{-4}
    Time class B = 2.66×10^{-4}
    Time class C = 10×10^{-4}
    Time class D = 5.33×10^{-4}

Total time P1 = 18.65×10^{-4}

P2: Time class A = 10^{-4}
    Time class B = 2×10^{-4}
    Time class C = 5×10^{-4}
    Time class D = 3×10^{-4}

Total time P2 = 11×10^{-4}

1.4.2   CPI = time $\times$ clock rate/No. instr.

$$CPI(P1) = 18.65 \times 10^{-4} \times 1.5 \times 10^9/10^6 = 2.79$$
$$CPI(P2) = 11 \times 10^{-4} \times 2 \times 10^9/10^6 = 2.2$$

1.4.3

$$\text{clock cycles(P1)} = 10^5 \times 1 + 2 \times 10^5 \times 2 + 5 \times 10^5 \times 3 + 2 \times 10^5 \times 4 = 28 \times 10^5$$
$$\text{clock cycles(P2)} = 10^5 \times 2 + 2 \times 10^5 \times 2 + 5 \times 10^5 \times 2 + 2 \times 10^5 \times 3 = 22 \times 10^5$$

1.4.4

$(500 \times 1 + 50 \times 5 + 100 \times 5 + 50 \times 2) \times 0.5^{-9} = 675$ ns

1.4.5   CPI = time $\times$ clock rate/No. instr.

$CPI = 675 \times 10^{-9} \times 2 \times 10^9/700 = 1.92$

1.4.6

$Time = (500 \times 1 + 50 \times 5 + 50 \times 5 + 50 \times 2) \times 0.5^{-9} = 550$ ns

Speed-up = 675 ns/550 ns = 1.22

$CPI = 550 \times 10^{-9} \times 2 \times 10^8/700 = 1.57$

Solution 1.5

1.5.1

a.	1G, 0.75G inst/s
b.	1G, 1.5G inst/s

1.5.2

a.	P2 is 1.33 times faster than P1
b.	P1 is 1.03 times faster than P2

1.5.3

a.	P2 is 1.31 times faster than P1
b.	P1 is 1.00 times faster than P2

1.5.4

a.	2.05 μs
b.	1.93 μs

1.5.5

a.	0.71 μs
b.	0.86 μs

1.5.6

a.	1.30 times faster
b.	1.40 times faster

Solution 1.6

1.6.1

	Compiler A CPI	Compiler B CPI
a.	1.00	1.17
b.	0.80	0.58

1.6.2

a.	0.86
b.	1.37

1.6.3

	Compiler A speed-up	Compiler B speed-up
a.	1.52	1.77
b.	1.21	0.88

1.6.4

	P1 peak	P2 peak
a.	4G Inst/s	3G Inst/s
b.	4G Inst/s	3G Inst/s

1.6.5   Speed-up, P1 versus P2:

a.	0.967105263
b.	0.730263158

1.6.6

a.	6.204081633
b.	8.216216216

Solution 1.7

1.7.1

Geometric mean clock rate ratio $= (1.28 \times 1.56 \times 2.64 \times 3.03 \times 10.00 \times 1.80 \times 0.74)^{1/7} = 2.15$
Geometric mean power ratio $= (1.24 \times 1.20 \times 2.06 \times 2.88 \times 2.59 \times 1.37 \times 0.92)^{1/7} = 1.62$

1.7.2

Largest clock rate ratio $=$ 2000 MHz/200 MHz $=$ 10 (Pentium Pro to Pentium 4 Willamette)
Largest power ratio = 29.1 W/10.1 W = 2.88 (Pentium to Pentium Pro)

1.7.3

Clock rate: $2.667 \times 10^9 / 12.5 \times 10^6 = 212.8$
Power: 95 W/3.3 W $= 28.78$

1.7.4 $C = P/V^2 \times$ clockrate

80286: $C = 0.0105 \times 10^{-6}$
80386: $C = 0.01025 \times 10^{-6}$
80486: $C = 0.00784 \times 10^{-6}$
Pentium: $C = 0.00612 \times 10^{-6}$
Pentium Pro: $C = 0.0133 \times 10^{-6}$
Pentium 4 Willamette: $C = 0.0122 \times 10^{-6}$
Pentium 4 Prescott: $C = 0.00183 \times 10^{-6}$
Core 2: $C = 0.0294 \times 10^{-6}$

1.7.5 $3.3/1.75 = 1.78$ (Pentium Pro to Pentium 4 Willamette)

1.7.6

Pentium to Pentium Pro: $3.3/5 = 0.66$
Pentium Pro to Pentium 4 Willamette: $1.75/3.3 = 0.53$
Pentium 4 Willamette to Pentium 4 Prescott: $1.25/1.75 = 0.71$
Pentium 4 Prescott to Core 2: $1.1/1.25 = 0.88$
Geometric mean $= 0.68$

Solution 1.8

1.8.1 $\text{Power}_1 = V^2 \times$ clock rate $\times$ C. $\text{Power}_2 = 0.9\ \text{Power}_1$

$C_2/C_1 = 0.9 \times 5^2 \times 0.5 \times 10^9 / 3.3^2 \times 1 \times 10^9 = 1.03$

1.8.2 $\text{Power}_2/\text{Power}_1 = V_2^2 \times$ clock rate$_2 / V_1^2 \times$ clock rate$_1$

$\text{Power}_2/\text{Power}_1 = 0.87 \Rightarrow$ Reduction of 13%

1.8.3

$\text{Power}_2 = V_2^2 \times 1 \times 10^9 \times 0.8 \times C_1 = 0.6 \times \text{Power}_1$
$\text{Power}_1 = 5^2 \times 0.5 \times 10^9 \times C_1$
$V_2^2 \times 1 \times 10^9 \times 0.8 \times C_1 = 0.6 \times 5^2 \times 0.5 \times 10^9 \times C_1$
$V_2 = ((0.6 \times 5^2 \times 0.5 \times 10^9)/(1 \times 10^9 \times 0.8))^{1/2} = 3.06$ V

1.8.4 $Power_{new} = 1 \times C_{old} \times V^2_{old}/(2^{-1/4})^2 \times$ clock rate $\times 2^{1/2} = Power_{old}$. Thus, power scales by 1.

1.8.5 $1/2^{-1/2} = 2^{1/2}$

1.8.6 Voltage $= 1.1 \times 1/2^{-1/4} = 0.92$ V. Clock rate $= 2.667 \times 2^{1/2} = 3.771$ GHz

Solution 1.9

1.9.1

a.	$1/49 \times 100 = 2\%$
b.	$45/120 \times 100 = 37.5\%$

1.9.2

a.	$I_{leak} = 1/3.3 = 0.3$
b.	$I_{leak} = 45/1.1 = 40.9$

1.9.3

a.	$Power_{st}/Power_{dyn} = 1/49 = 0.02$
b.	$Power_{st}/Power_{dyn} = 45/57 = 0.6$

1.9.4 $Power_{st}/Power_{dyn} = 0.6 \Rightarrow Power_{st} = 0.6 \times Power_{dyn}$

a.	$Power_{st} = 0.6 \times 40$ W $= 24$ W
b.	$Power_{st} = 0.6 \times 30$ W $= 18$ W

1.9.5

a.	$I_{lk} = 24/0.8 = 30$ A
b.	$I_{lk} = 18/0.8 = 22.5$ A

1.9.6

	Power $_{st}$ at 1.0 V	I_{lk} at 1.0 V	Power $_{st}$ at 1.2 V	I_{lk} at 1.2 V	Larger
a.	119 W	119 A	136 W	113.3 A	I_{lk} at 1.0 V
b.	93.5 W	93.5 A	110.5 W	92.1 A	I_{lk} at 1.0 V

Solution 1.10

1.10.1

a.	Processors	Instructions per processor	Total instructions
	1	4096	4096
	2	2048	4096
	4	1024	4096
	8	512	4096

b.	Processors	Instructions per processor	Total instructions
	1	4096	4096
	2	2278	4556
	4	1464	5856
	8	1132	9056

1.10.2

a.	Processors	Execution time (μs)
	1	4.096
	2	2.048
	4	1.024
	8	0.512

b.	Processors	Execution time (μs)
	1	4.096
	2	3.203
	4	3.164
	8	3.582

1.10.3

a.	Processors	Execution time (μ s)
	1	5.376
	2	2.688
	4	1.344
	8	0.672

b.	Processors	Execution time (μ s)
	1	5.376
	2	3.878
	4	3.564
	8	3.882

1.10.4

a.	Cores	Execution time (s) @ 3 GHz
	1	4.00
	2	2.17
	4	1.25
	8	0.75

b.	Cores	Execution time (s) @ 3 GHz
	1	4.00
	2	2.00
	4	1.00
	8	0.50

1.10.5

a.	Cores	Power (W) per core @ 3 GHz	Power (W) per core @ 500 MHz	Power (W) @ 3 GHz	Power (W) @ 500 MHz
	1	15	0.625	15	0.625
	2	15	0.625	30	1.25
	4	15	0.625	60	2.5
	8	15	0.625	120	5

b.	Cores	Power (W) per core @ 3 GHz	Power (W) per core @ 500 MHz	Power (W) @ 3 GHz	Power (W) @ 500 MHz
	1	15	0.625	15	0.625
	2	15	0.625	30	1.25
	4	15	0.625	60	2.5
	8	15	0.625	120	5

1.10.6

a.	Processors	Energy (J) @ 3 GHz	Energy (J) @ 500 MHz
	1	60	15
	2	65	16.25
	4	75	18.75
	8	90	22.5

b.	Processors	Energy (J) @ 3 GHz	Energy (J) @ 500 MHz
	1	60	15
	2	60	15
	4	60	15
	8	60	15

Solution 1.11

1.11.1   Wafer area $= \pi \times (d/2)^2$

a.	Wafer area $= \pi \times 7.5^2 = 176.7$ cm^2
b.	Wafer area $= \pi \times 12.5^2 = 490.9$ cm^2

Die area $=$ wafer area/dies per wafer

a.	Die area $= 176.7/90 = 1.96$ cm^2
b.	Die area $= 490.9/140 = 3.51$ cm^2

Yield $= 1/(1 + ($defect per area $\times$ die area$)/2)^2$

a.	Yield $= 0.97$
b.	Yield $= 0.92$

1.11.2   Cost per die $=$ cost per wafer/(dies per wafer $\times$ yield)

a.	Cost per die $= 0.12$
b.	Cost per die $= 0.16$

1.11.3

a.	Dies per wafer $= 1.1 \times 90 = 99$ Defects per area $= 1.15 \times 0.018 = 0.021$ defects/cm^2 Die area $=$ wafer area/Dies per wafer $= 176.7/99 = 1.78$ cm^2 Yield $= 0.97$
b.	Dies per wafer $= 1.1 \times 140 = 154$ Defects per area $= 1.15 \times 0.024 = 0.028$ defects/cm^2 Die area $=$ wafer area/Dies per wafer $= 490.9/154 = 3.19$ cm^2 Yield $= 0.93$

1.11.4   Yield $= 1/(1 + ($defect per area $\times$ die area$)/2)^2$

Then defect per area $= (2/$die area$)(y^{-1/2} - 1)$

Replacing values for T1 and T2 we get

T1: defects per area$= 0.00085$ defects/mm$^2 = 0.085$ defects/cm^2
T2: defects per area$= 0.00060$ defects/mm$^2 = 0.060$ defects/cm^2
T3: defects per area$= 0.00043$ defects/mm$^2 = 0.043$ defects/cm^2
T4: defects per area$= 0.00026$ defects/mm$^2 = 0.026$ defects/cm^2

1.11.5   no solution provided

Solution 1.12

1.12.1   CPI = clock rate $\times$ CPU time/instr. count

clock rate = 1/cycle time = 3 GHz

a.	CPI(pearl) = 3 $\times$ $10^9 \times$ 500/2118 $\times$ 10^9 = 0.7
b.	CPI(mcf) = 3 $\times$ $10^9 \times$ 1200/336 $\times$ 10^9 = 10.7

1.12.2   SPECratio = ref. time/execution time.

a.	SPECratio(pearl) = 9770/500 = 19.54
b.	SPECratio(mcf) = 9120/1200 = 7.6

1.12.3

(19.54 $\times$ 7.6)$^{1/2}$ = 12.19

1.12.4   CPU time = No. instr. $\times$ 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%.

1.12.5   CPU time(before) = No. instr. $\times$ CPI/clock rate

CPU time(after) = 1.1 $\times$ No. instr. $\times$ 1.05 $\times$ CPI/clock rate
CPU times(after)/CPU time(before) = 1.1 $\times$ 1.05 = 1.155. Thus, CPU time is increased by 15.5%

1.12.6   SPECratio = reference time/CPU time

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

Solution 1.13

1.13.1   CPI = (CPU time $\times$ clock rate)/No. instr.

a.	CPI = 450 $\times$ 4 $\times$ 10^9/(0.85 $\times$ 2118 $\times$ 10^8) = 0.99
b.	CPI = 1150 $\times$ 4 $\times$ 10^9/(0.85 $\times$ 336 $\times$ 10^8) = 16.10

1.13.2   Clock rate ratio = 4 GHz/3 GHz = 1.33.

a.	CPI @ 4 GHz = 0.99, CPI @ 3 GHz = 0.7, ratio = 1.41
b.	CPI @ 4 GHz = 16.1, CPI @ 3 GHz = 10.7, ratio = 1.50

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

1.13.3

a.	450/500 = 0.90. CPU time reduction: 10%.
b.	1150/1200 = 0.958. CPU time reduction: 4.2%.

1.13.4   No. instr. = CPU time $\times$ clock rate/CPI.

a.	No. instr. = 820 $\times$ 0.9 $\times$ 4 $\times 10^9/0.96 = 3075 \times 10^9$
b.	No. instr. = 580 $\times$ 0.9 $\times$ 4 $\times 10^9/2.94 = 710 \times 10^9$

1.13.5   Clock rate = No. instr. $\times$ CPI/CPU time.

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

1.13.6   Clock rate = No. instr. $\times$ CPI/CPU time.

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

Solution 1.14

1.14.1   No. instr. = 10^6

T_{cpu} (P1) = $10^6 \times$ 1.25/4 $\times 10^9 = 0.315 \times 10^{-3}$s
T_{cpu} (P2) = $10^6 \times$ 0.75/3 $\times 10^9 = 0.25 \times 10^{-3}$s
clock rate(P1) > clock rate(P2), but performance(P1) < performance(P2)

1.14.2

P1: 10^6 instructions, T_{cpu}(P1) = 0.315 $\times 10^{-3}$s
P2: T_{cpu}(P2) = N $\times$ 0.75/3 $\times 10^9$, then N = 1.26 $\times 10^6$

1.14.3   MIPS $= $ Clock rate $\times 10^{-6}$/CPI

MIPS(P1) $= 4 \times 10^9 \times 10^{-6}$/1.25 $= 3200$

MIPS(P2) $= 3 \times 10^9 \times 10^{-6}$/0.75 $= 4000$

MIPS(P1) < MIPS(P2), performance(P1) < performance(P2) in this case (from 1.14.1)

1.14.4

a.	FP op $= 10^6 \times 0.4 = 4 \times 10^5$, clock cyles $_{fp} =$ CPI $\times$ No. FP instr. $= 4 \times 10^5$ $T_{fp} = 4 \times 10^5 \times 0.33 \times 10^9 = 1.32 \times 10^4$ then MFLOPS $= 3.03 \times 10^3$
b.	FP op $= 3 \times 10^6 \times 0.4 = 1.2 \times 10^6$, clock cyles $_{fp} =$ CPI $\times$ No. FP instr. $= 0.70 \times 1.2 \times 10^6$ $T_{fp} = 0.84 \times 10^6 \times 0.33 \times 10^9 = 2.77 \times 10^4$ then MFLOPS $= 4.33 \times 10^3$

1.14.5   CPU clock cycles $=$ FP cycles $+$ CPI(L/S) $\times$ No. instr. (L/S) + CPI(Branch) $\times$ No. instr. (Branch)

a.	5×10^5 L/S instr., 4×10^5 FP instr. and 10^5 Branch instr. CPU clock cycles $= 4 \times 10^5 + 0.75 \times 5 \times 10^5 + 1.5 \times 10^5 = 9.25 \times 10^5$ $T_{cpu} = 9.25 \times 10^5 \times 0.33 \times 10^9 = 3.05 \times 10^4$ MIPS $= 10^6/(3.05 \times 10^4 \times 10^6) = 3.2 \times 10^1$
b.	1.2×10^6 L/S instr., 1.2×10^6 FP instr. and 0.6×10^6 Branch instr. CPU clock cycles $= 0.84 \times 10^6 + 1.25 \times 1.2 \times 10^6 + 1.25 \times 0.6 \times 10^6 = 3.09 \times 10^6$ $T_{cpu} = 3.09 \times 10^6 \times 0.33 \times 10^9 = 1.01 \times 10^3$ MIPS $= 3 \times 10^6/(1.01 \times 10^{-3} \times 10^6) = 2.97 \times 10^3$

1.14.6

a.	performance $= 1/T_{cpu} = 3.2 \times 10^3$
b.	performance $= 1/T_{cpu} = 9.9 \times 10^2$

The second program has the higher performance and the higher MFLOPS ? gure, but the ? rst program has the higher MIPS ? gure.

Solution 1.15

1.15.1

a.	$T_{fp} = 35 \times 0.8 = 28$ s, $T_{p1} = 28 + 85 + 50 + 30 = 193$ s. Reduction: 3.5%
b.	$T_{fp} = 50 \times 0.8 = 40$ s, $T_{p4} = 40 + 80 + 50 + 30 = 200$ s. Reduction: 4.7%

1.15.2

a.	T_{p1} = 200 × 0.8 = 160 s, T_{fp} + $T_{l/s}$ + T_{branch} = 115 s, T_{int} = 45 s. Reduction time INT: 47%
b.	T_{p4} = 210 × 0.8 = 168 s, T_{fp} + $T_{l/s}$ + T_{branch} = 130 s, T_{int} = 38 s. Reduction time INT: 52.4%

1.15.3

a.	T_{p1} = 200 × 0.8 = 160 s, T_{fp} + T_{int} + $T_{l/s}$ = 170 s. NO
b.	T_{p4} = 210 × 0.8 = 168 s, T_{fp} + T_{int} + $T_{l/s}$ = 180 s. NO

1.15.4

Clock cyles = 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

a.	1 processor: clock cycles = 8192; T_{cpu} = 4.096 s
b.	8 processors: clock cycles = 1024; T_{cpu} = 0.512 s

To half 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.

a.	1 processor: $CPI_{improved\ fp}$ = (4096 – 7632)/560 < 0 ==> not possible
b.	8 processors: $CPI_{improved\ fp}$ = (512 – 944)/80 < 0 ==> not possible

1.15.5 Using the clock cycle data from 1.15.4:

To half 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.

a.	1 processor: CPI $_{\text{improved l/s}}$ = (4096 – 3072)/1280 = 0.8
b.	8 processors: CPI $_{\text{improved l/s}}$ = (512 – 384)/160 = 0.8

1.15.6

Clock cyles $= \text{CPI}_{\text{fp}} \times$ No. FP instr. $+ \text{CPI}_{\text{int}} \times$ No. INT instr. $+ \text{CPI}_{\text{l/s}} \times$ No. L/S instr. $+ \text{CPI}_{\text{branch}} \times$ No. branch instr.

$T_{\text{cpu}} =$ clock cycles/clock rate = clock cycles/2×10^{9}

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

a.	1 processor: T $_{\text{cpu}}$(before improv.) = 4.096 s; T $_{\text{cpu}}$(after improv.) = 2.739 s
b.	8 processors: T $_{\text{cpu}}$ (before improv.) = 0.512 s; T $_{\text{cpu}}$(after improv.) = 0.342 s

Solution 1.16

1.16.1   Without reduction in any routine:

a.	total time 2 proc = 185 ns
b.	total time 16 proc = 34 ns

Reducing time in routines A, C and E:

a.	2 proc: T(A) = 17 ns, T(C) = 8.5 ns, T(E) = 4.1 ns, total time = 179.6 ns ==> reduction = 2.9%
b.	16 proc: T(A) = 3.4 ns, T(C) = 1.7 ns, T(E) = 1.7 ns, total time = 32.8 ns ==> reduction = 3.5%

1.16.2

a.	2 proc: T(B) = 72 ns, total time = 177 ns ==> reduction = 4.3%
b.	16 proc: T(B) = 12.6 ns, total time = 32.6 ns ==> reduction = 4.1%

1.16.3

a.	2 proc: T(D) = 63 ns, total time = 178 ns ==> reduction = 3.7%
b.	16 proc: T(D) = 10.8 ns, total time = 32.8 ns ==> reduction = 3.5%

1.16.4

# Processors	Computing time	Computing time ratio	Routing time ratio
2	176		
4	96	0.55	1.18
8	49	0.51	1.31
16	30	0.61	1.29
32	14	0.47	1.05
64	6.5	0.46	1.13

1.16.5 Geometric mean of computing time ratios = 0.52. Multiply this by the computing time for a 64-processor system gives a computing time for a 128-processor system of 3.4 ms.

Geometric mean of routing time ratios = 1.19. Multiply this by the routing time for a 64-processor system gives a routing time for a 128-processor system of 30.9 ms.

1.16.6 Computing time = 176/0.52 = 338 ms. Routing time = 0, since no communication is required.

2 Solutions

Solution 2.1

2.1.1

a.	add f, g, h
	add f, f, i
	add f, f, j

b.	addi f, h, 5 addi f, f, g

2.1.2

a.	3
b.	2

2.1.3

a.	14
b.	10

2.1.4

a.	f = g + h
b.	f = g + h

2.1.5

a.	5
b.	5

Solution 2.2

2.2.1

a.	add f, f, f add f, f, i
b.	addi f, j, 2 add f, f, g

2.2.2

a.	2
b.	2

2.2.3

a.	6
b.	5

2.2.4

a.	f += h;
b.	f = 1 – f;

2.2.5

a.	4
b.	0

Solution 2.3

2.3.1

a.	add f, f, g
	add f, f, h
	add f, f, i
	add f, f, j
	addi f, f, 2

b.	addi f, f, 5 sub f, g, f

2.3.2

a.	5
b.	2

2.3.3

a.	17
b.	– 4

2.3.4

a.	f = h – g;
b.	f = g – f – 1;

2.3.5

a.	1
b.	0

Solution 2.4

2.4.1

a.	lw $s0, 16($s7)
	add $s0, $s0, $s1
	add $s0, $s0, $s2

b.	lw $t0, 16($s7)
	lw $s0, 0($t0)
	sub $s0, $s1, $s0

2.4.2

a.	3
b.	3

2.4.3

a.	4
b.	4

2.4.4

a.	f += g + h + i + j;
b.	f = A[1];

2.4.5

a.	no change
b.	no change

2.4.6

a.	5 as written, 5 minimally
b.	2 as written, 2 minimally

Solution 2.5

2.5.1

a.	Address Data		temp = Array[3]; Array[3] = Array[2]; Array[2] = Array[1]; Array[1] = Array[0]; Array[0] = temp;
	12	1	
	8	6	
	4	4	
	0	2	

b.	Address Data		temp = Array[4];
			Array[4] = Array[0];
			Array[0] = temp;
			temp = Array[3];
			Array[3] = Array[1];
			Array[1] = temp;

	16	1	
	12	2	
	8	3	
	4	4	
	0	5	

2.5.2

a.	Address Data		temp = Array[3];
			Array[3] = Array[2];
			Array[2] = Array[1];
			Array[1] = Array[0];
			Array[0] = temp;
				lw $t0, 12($s6)
				lw $t1, 8($s6)
				sw $t1, 12($s6)
				lw $t1, 4($s6)
				sw $t1, 8($s6)
				lw $t1, 0($s6)
				sw $t1, 4($s6)
				sw $t0, 0($s6)

	12	1		
	8	6		
	4	4		
	0	2		

b.	Address Data		temp = Array[4];
			Array[4] = Array[0];
			Array[0] = temp;
			
			temp = Array[3];
			Array[3] = Array[1];
			Array[1] = temp;
				lw $t0, 16($s6)
				lw $t1, 0($s6)
				sw $t1, 16($s6)
				sw $t0, 0($s6)
				
				lw $t0, 12($s6)
				lw $t1, 4($s6)
				sw $t1, 12($s6)
				sw $t0, 4($s6)

	16	1		
	12	2		
	8	3		
	4	4		
	0	5		

2.5.3

a.	Address
	12
	8
	4
	0
		Data
		1
		6
		4
		2
			temp = Array[3];
			Array[3] = Array[2];
			Array[2] = Array[1];
			Array[1] = Array[0];
			Array[0] = temp;
				lw $t0, 12($s6)
				lw $t1, 8($s6)
				sw $t1, 12($s6)
				lw $t1, 4($s6)
				sw $t1, 8($s6)
				lw $t1, 0($s6)
				sw $t1, 4($s6)
				sw $t0, 0($s6)
					8 mips instructions, +1 mips inst. for every non-zero offset lw/sw pair (11 mips inst.)

b.	Address
	16
	12
	8
	4
	0
		Data
		1
		2
		3
		4
		5
			temp = Array[4];
			Array[4] = Array[0];
			Array[0] = temp;
			
			temp = Array[3];
			Array[3] = Array[1];
			Array[1] = temp;
				lw $t0, 16($s6)
				lw $t1, 0($s6)
				sw $t1, 16($s6)
				sw $t0, 0($s6)
				
				lw $t0, 12($s6)
				lw $t1, 4($s6)
				sw $t1, 12($s6)
				sw $t0, 4($s6)
					8 mips instructions, +1 mips inst. for every non-zero offset lw/sw pair (11 mips inst.)

2.5.4

a.	305419896
b.	3199070221

2.5.5

	Little-Endian		Big-Endian	
a.	Address 12 8 4 0	Data 12 34 56 78	Address 12 8 4 0	Data 78 56 34 12
b.	Address 12 8 4 0	Data be ad f0 0d	Address 12 8 4 0	Data 0d f0 ad be

Solution 2.6

2.6.1

a.	lw $s0, 4($s7)
	sub $s0, $s0, $s1
	add $s0, $s0, $s2

b.	add $t0, $s7, $s1
	lw $t0, 0($t0)
	add $t0, $t0, $s6
	lw $s0, 4($t0)

2.6.2

a.	3
b.	4

2.6.3

a.	4
b.	5

2.6.4

a.	f = 2i + h;
b.	f = A[g – 3];

2.6.5

a.	$s0 = 110
b.	$s0 = 300

2.6.6

a.

	Type	opcode	rs	rt	rd	immed
add $s0, $s0, $s1	R-type	0	16	17	16	
add $s0, $s3, $s2	R-type	0	19	18	16	
add $s0, $s0, $s3	R-type	0	16	19	16	

b.

	Type	opcode	rs	rt	rd	immed
addi $s6, $s6, – 20	I-type	8	22	22		– 20
add $s6, $s6, $s1	R-type	0	22q	17	22	
lw $s0, 8($s6)	I-type	35	22	16		8

Solution 2.7

2.7.1

a.	− 1391460350
b.	− 19629

2.7.2

a.	2903506946
b.	4294947667

2.7.3

a.	AD100002
b.	FFFFB353

2.7.4

a.	011111111111111111111111111111111
b.	1111101000

2.7.5

a.	7FFFFFFF
b.	3E8

2.7.6

a.	80000001
b.	FFFFFC18

Solution 2.8

2.8.1

a.	7FFFFFFF, no over? ow
b.	80000000, over? ow

2.8.2

a.	60000001, no over? ow
b.	0, no over? ow

2.8.3

a.	EFFFFFFF, over? ow
b.	C0000000, over? ow

2.8.4

a.	over?ow
b.	no over?ow

2.8.5

a.	no over?ow
b.	no over?ow

2.8.6

a.	over?ow
b.	no over?ow

Solution 2.9

2.9.1

a.	over?ow
b.	no over?ow

2.9.2

a.	over?ow
b.	no over?ow

2.9.3

a.	no over? ow
b.	over? ow

2.9.4

a.	no over? ow
b.	no over? ow

2.9.5

a.	1D100002
b.	6FFFB353

2.9.6

a.	487587842
b.	1879028563

Solution 2.10

2.10.1

a.	sw $t3, 4($s0)
b.	lw $t0, 64($t0)

2.10.2

a.	I-type
b.	I-type

2.10.3

a.	AE0B0004
b.	8D080040

2.10.4

a.	0x01004020
b.	0x8E690004

2.10.5

a.	R-type
b.	I-type

2.10.6

a.	op=0x0, rd=0x8, rs=0x8, rt=0x0, funct=0x0
b.	op=0x23, rs=0x13, rt=0x9, imm=0x4

Solution 2.11

2.11.1

a.	1010 1110 0000 1011 1111 1111 1111 1100 $_{two}$
b.	1000 1101 0000 1000 1111 1111 1100 0000 $_{two}$

2.11.2

a.	2920022012
b.	2366177216

2.11.3

a.	sw $t3, – 4($s0)
b.	lw $t0, – 64($t0)

2.11.4

a.	R-type
b.	I-type

2.11.5

a.	add $v1, $at, $v0
b.	sw $a1, 4($s0)

2.11.6

a.	0x00221820
b.	0xAD450004

Solution 2.12

2.12.1

	Type	opcode	rs	rt	rd	shamt	funct	
a.	R-type	6	3	3	3	5	6	total bits = 26
b.	R-type	6	5	5	5	5	6	total bits = 32

2.12.2

	Type	opcode	rs	rt	immed	
a.	I-type	6	3	3	16	total bits = 28
b.	I-type	6	5	5	10	total bits = 26

2.12.3

a.	less registers less bits per instruction could reduce code size less registers more register spills more instructions
b.	smaller constants more lui instructions could increase code size smaller constants smaller opcodes smaller code size

2.12.4

a.	17367056
b.	2366177298

2.12.5

a.	add $t0, $t1, $0
b.	lw $t1, 12($t0)

2.12.6

a.	R-type, op=0 $\times$ 0, rt=0 $\times$ 9
b.	I-type, op=0 $\times$ 23, rt=0 $\times$ 8

Solution 2.13

2.13.1

a.	0x57755778
b.	0xFEFFFEDE

2.13.2

a.	0x55555550
b.	0xEADFEED0

2.13.3

a.	0x0000AAAA
b.	0x0000BFCD

2.13.4

a.	0x00015B5A
b.	0x00000000

2.13.5

a.	0x5b5a0000
b.	0x000000f0

2.13.6

a.	0xEFEFFFFF
b.	0x000000F0

Solution 2.14

2.14.1

a.	add $t1, $t0, $0 srl $t1, $t1, 5 andi $t1, $t1, 0x0001ffff
b.	add $t1, $t0, $0 sll $t1, $t1, 10 andi $t1, $t1, 0xffff8000

2.14.2

a.	add $t1, $t0, $0 andi $t1, $t1, 0x0000000f
b.	add $t1, $t0, $0 srl $t1, $t1, 14 andi $t1, $t1, 0x0003c000

2.14.3

a.	add $t1, $t0, $0 srl $t1, $t1, 28
b.	add $t1, $t0, $0 srl $t1, $t1, 14 andi $t1, $t1, 0x0001c000

2.14.4

a.	add $t2, $t0, $0 srl $t2, $t2, 11 and $t2, $t2, 0x0000003f and $t1, $t1, 0xffffffc0 ori $t1, $t1, $t2
b.	add $t2, $t0, $0 sll $t2, $t2, 3 and $t2, $t2, 0x000fc000 and $t1, $t1, 0xfff03fff ori $t1, $t1, $t2

2.14.5

a.	add $t2, $t0, $0 and $t2, $t2, 0x0000001f and $t1, $t1, 0xffffffe0 ori $t1, $t1, $t2
b.	add $t2, $t0, $0 sll $t2, $t2, 14 and $t2, $t2, 0x0007c000 and $t1, $t1, 0xfff83fff ori $t1, $t1, $t2

2.14.6

a.	add $t2, $t0, $0 srl $t2, $t2, 29 and $t2, $t2, 0x00000003 and $t1, $t1, 0xfffffffc ori $t1, $t1, $t2
b.	add $t2, $t0, $0 srl $t2, $t2, 15 and $t2, $t2, 0x0000c000 and $t1, $t1, 0xffff3fff ori $t1, $t1, $t2

Solution 2.15

2.15.1

a.	0x0000a581
b.	0x00ff5a66

2.15.2

a.	nor $t1, $t2, $t2 and $t1, $t1, $t3
b.	xor $t1, $t2, $t3 nor $t1, $t1, $t1

2.15.3

a.	nor $t1, $t2, $t2 and $t1, $t1, $t3	000000 01010 01010 01001 00000 100111 000000 01001 01011 01001 00000 100100
b.	xor $t1, $t2, $t3 nor $t1, $t1, $t1	000000 01010 01011 01001 00000 100110 000000 01001 01001 01001 00000 100111

2.15.4

a.	0x00000220
b.	0x00001234

2.15.5   Assuming $t1 = A, $t2 = B, $s1= base of Array C

a.	lw $t3, 0($s1) and $t1, $t2, $t3
b.	beq $t1, $0, ELSE add $t1, $t2, $0 beq $0, $0, END ELSE: lw $t2, 0($s1) END:

2.15.6

a.	lw $t3, 0($s1) and $t1, $t2, $t3	100011 10001 01011 0000000000000000 000000 01010 01011 01001 00000 100100
b.	beq $t1, $0, ELSE add $t1, $t2, $0 beq $0, $0, END ELSE: lw $t2, 0($s1) END:	000100 01001 00000 0000000000000010 000000 01010 00000 01001 00000 100000 000100 00000 00000 0000000000000001 100011 10001 01010 0000000000000000

Solution 2.16

2.16.1

a.	$t2 = 1
b.	$t2 = 1

2.16.2

a.	all, 0x8000 to 0x7FFFF
b.	0x8000 to 0xFFFE

2.16.3

a.	jump—no, beq —no
b.	jump—no, beq —no

2.16.4

a.	$t2 = 2
b.	$t2 = 2

2.16.5

a.	$t2 = 0
b.	$t2 = 1

2.16.6

a.	jump—yes, beq —no
b.	jump—yes, beq —yes

Solution 2.17

2.17.1   The answer is really the same for all. All of these instructions are either supported by an existing instruction, or sequence of existing instructions. Looking for an answer along the lines of,   " these instructions are not common, and we are only making the common case fast .   "

2.17.2

a.	could be either R-type of I-type
b.	R-type

2.17.3

a.	ABS: sub $t2,$zero,$t3 # t2 = – t3
	        ble $t3,$zero,done # if t3 < 0, result is t2
	        add $t2,$t3,$zero # if t3 > 0, result is t3
	DONE:

b.	slt $t1, $t3, $t2

2.17.4

a.	20
b.	200

2.17.5

a.	i = 10;
	do {
	   B += 2;
	   i = i – 1;
	} while (i > 0)

b.	i = 10;
	do {
	   temp = 10;
	   do {
	      B += 2;
	      temp = temp – 1;
	   } while (temp > 0)
	   i = i – 1;
	} while (i > 0)

2.17.6

a.	$5 \times N + 3$
b.	$33 \times N$

Solution 2.18

2.18.1

a.	
b.	

2.18.2

a.	addi $t0, $0, 0
	    beq $0, $0, TEST
	LOOP: add $s0, $s0, $s1
	    addi $t0, $t0, 1
	TEST: slti $t2, $t0, 10
	    bne $t2, $0, LOOP

b.	LOOP: slti $t2, $s0, 10
	    beq $t2, $0, DONE
	    add $t3, $s1, $s0
	    sll $t2, $s0, 2
	    add $t2, $s2, $t2
	    sw $t3, ($t2)
	    addi $s0, $s0, 1
	    j LOOP
	DONE:

2.18.3

a.	6 instructions to implement and 44 instructions executed
b.	8 instructions to implement and 2 instructions executed

2.18.4

a.	501
b.	301

2.18.5

a.	for(i=100; i>0; i --){
	   result += MemArray[s0];
	   s0 += 1;
	}

b.	for(i=0; i<100; i+=2){
	   result += MemArray[s0 + i];
	   result += MemArray[s0 + i + 1];
	
	}

2.18.6

a.	addi $t1, $s0, 400
	LOOP: lw $s1, 0($s0)
	    add $s2, $s2, $s1
	    addi $s0, $s0, 4
	    bne $s0, $t1, LOOP

b.	already reduced to minimum instructions

Solution 2.19

2.19.1

| a. | ```
compare:
    addi $sp, $sp,      – 4
    sw  $ra, 0($sp)

    add  $s0, $a0, $0
    add  $s1, $a1, $0

    jal  sub
    addi $t1, $0, 1
    beq  $v0, $0, exit
    slt  $t2, $0, $v0
    bne  $t2, $0, exit
    addi $t1, $0, $0
exit:
    add  $v0, $t1, $0
    lw   $ra, 0($sp)
    addi $sp, $sp, 4
    jr   $ra

sub:
    sub  $v0, $a0, $a1
    jr   $ra
``` |
| b. | ```
?b_iter:
    addi $sp, $sp,      – 16
    sw   $ra, 12($sp)
    sw   $s0, 8($sp)
    sw   $s1, 4($sp)
    sw   $s2, 0($sp)

    add  $s0, $a0, $0
    add  $s1, $a1, $0
    add  $s2, $a2, $0

    add  $v0, $s1, $0,
    bne  $s2, $0, exit

    add  $a0, $s0, $s1
    add  $a1, $s0, $0
    add  $a2, $s2,      – 1
    jal  ?       b_iter
exit:
    lw   $s2, 0($sp)
    lw   $s1, 4($sp)
    lw   $s0, 8($sp)
    lw   $ra, 12($sp)
    addi $sp, $sp, 16
    jr   $ra
``` |

2.19.2

<table>
<tr><td>a.</td><td>

```
compare:
    addi $sp, $sp,        – 4
    sw   $ra, 0($sp)

    sub  $t0, $a0, $a1
    addi $t1, $0, 1
    beq  $t0, $0, exit
    slt  $t2, $0, $t0
    bne  $t2, $0, exit
    addi $t1, $0, $0
exit:
    add  $v0, $t1, $0
    lw   $ra, 0($sp)
    addi $sp, $sp, 4
    jr   $ra
```
</td></tr>
<tr><td>b.</td><td>Due to the recursive nature of the code, not possible for the compiler to in-line the function call.</td></tr>
</table>

2.19.3

<table>
<tr><td>a.</td><td>

```
after calling function compare:
old $sp =>     0x7ffffffc    ???
$sp =>                – 4          contents of register $ra

after calling function sub:
old $sp =>     0x7ffffffc    ???
                      – 4          contents of register $ra
$sp =>                – 8                contents of register $ra   #return to
                                                          compare
```
</td></tr>
<tr><td>b.</td><td>

```
after calling function ?              b_iter:
old $sp =>     0x7ffffffc    ???
                      – 4          contents of register $ra
                      – 8          contents of register $s0
                      – 12           contents of register $s1
$sp =>                – 16           contents of register $s2
```
</td></tr>
</table>

2.19.4

<table>
<tr><td>a.</td><td>

```
f: addi    $sp,$sp,       – 8
    sw      $ra,4($sp)
    sw      $s0,0($sp)
    move    $s0,$a2
    jal     func
    move    $a0,$v0
    move    $a1,$s0
    jal     func
    lw      $ra,4($sp)
    lw      $s0,0($sp)
    addi    $sp,$sp,8
    jr      $ra
```
</td></tr>
</table>

b.	f: addi $sp,$sp, − 12
	sw $ra,8($sp)
	sw $s1,4($sp)
	sw $s0,0($sp)
	move $s0,$a1
	move $s1,$a2
	jal func
	move $a0,$s0
	move $a1,$s1
	move $s0,$v0
	jal func
	add $v0,$v0,$s0
	lw $ra,8($sp)
	lw $s1,4($sp)
	lw $s0,0($sp)
	addi $sp,$sp,12
	jr ra

2.19.5

a.	We can use the tail-call optimization for the second call to func , but then we must restore $ra and $sp before that call. We save only one instruction (jr $ra).
b.	We can NOT use the tail call optimization here, because the value returned from f is not equal to the value returned by the last call to func .

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

Solution 2.20

2.20.1

a.	FACT: addi $sp, $sp, − 8
	sw $ra, 4($sp)
	sw $a0, 0($sp)
	add $s0, $0, $a0
	slti $t0, $a0, 2
	beq $t0, $0, L1
	addi $v0, $0, 1
	addi $sp, $sp, 8
	jr $ra
	L1: addi $a0, $a0, − 1
	jal FACT
	mul $v0, $s0, $v0
	lw $a0, 0($sp)
	lw $ra, 4($sp)
	addi $sp, $sp, 8
	jr $ra

| b. | ```
FACT: addi  $sp, $sp,    – 8
      sw    $ra, 4($sp)
      sw    $a0, 0($sp)
      add   $s0, $0, $a0

      slti  $t0, $a0, 2
      beq   $t0, $0, L1

      addi  $v0, $0, 1
      addi  $sp, $sp, 8
      jr    $ra

L1:   addi  $a0, $a0,      – 1
      jal   FACT

      mul   $v0, $s0, $v0

      lw    $a0, 0($sp)
      lw    $ra, 4($sp)
      addi  $sp, $sp, 8
      jr    $ra
``` |

2.20.2

| a. | 25 MIPS instructions to execute nonrecursive vs. 45 instructions to execute (corrected version of) recursion

Nonrecursive version:

```
FACT:  addi  $sp, $sp,      – 4
       sw    $ra, 4($sp)
       add   $s0, $0, $a0
       add   $s2, $0, $1

LOOP:  slti  $t0, $s0, 2
       bne   $t0, $0, DONE
       mul   $s2, $s0, $s2
       addi  $s0, $s0,      – 1
       j LOOP

DONE:  add   $v0, $0, $s2
       lw    $ra, 4($sp)
       addi  $sp, $sp, 4
       jr    $ra
``` |
| b. | 25 MIPS instructions to execute nonrecursive vs. 45 instructions to execute (corrected version of) recursion

Nonrecursive version:

```
FACT:  addi  $sp, $sp,      – 4
       sw    $ra, 4($sp)
       add   $s0, $0, $a0
       add   $s2, $0, $1

LOOP:  slti  $t0, $s0, 2
       bne   $t0, $0, DONE
       mul   $s2, $s0, $s2
       addi  $s0, $s0,      – 1
       j LOOP

DONE:  add   $v0, $0, $s2
       lw    $ra, 4($sp)
       addi  $sp, $sp, 4
       jr    $ra
``` |

2.20.3

a.	Recursive version
	FACT: addi $sp, $sp, – 8
	      sw $ra, 4($sp)
	      sw $a0, 0($sp)
	      add $s0, $0, $a0
	HERE: slti $t0, $a0, 2
	      beq $t0, $0, L1
	      addi $v0, $0, 1
	      addi $sp, $sp, 8
	      jr $ra
	L1: addi $a0, $a0, – 1
	      jal FACT
	      mul $v0, $s0, $v0
	      lw $a0, 0($sp)
	      lw $ra, 4($sp)
	      addi $sp, $sp, 8
	      jr $ra

```
at label HERE, after calling function FACT with input of 4:
old $sp =>    0xnnnnnnnn    ???
                      – 4         contents of register $ra
$sp =>                – 8         contents of register $a0

at label HERE, after calling function FACT with input of 3:
old $sp =>    0xnnnnnnnn    ???
                      – 4         contents of register $ra
                      – 8         contents of register $a0
                      – 12        contents of register $ra
$sp =>                – 16        contents of register $a0

at label HERE, after calling function FACT with input of 2:
old $sp =>    0xnnnnnnnn    ???
                      – 4         contents of register $ra
                      – 8         contents of register $a0
                      – 12        contents of register $ra
                      – 16        contents of register $a0
                      – 20        contents of register $ra
$sp =>                – 24        contents of register $a0

at label HERE, after calling function FACT with input of 1:
old $sp =>    0xnnnnnnnn    ???
                      – 4         contents of register $ra
                      – 8         contents of register $a0
                      – 12        contents of register $ra
                      – 16        contents of register $a0
                      – 20        contents of register $ra
                      – 24        contents of register $a0
                      – 28        contents of register $ra
$sp =>                – 32        contents of register $a0
```

b. Recursive version

```
FACT:  addi  $sp, $sp,      – 8
       sw    $ra, 4($sp)
       sw    $a0, 0($sp)
       add   $s0, $0, $a0
HERE:  slti  $t0, $a0, 2
       beq   $t0, $0, L1
       addi  $v0, $0, 1
       addi  $sp, $sp, 8
       jr    $ra
L1:    addi  $a0, $a0,      – 1
       jal   FACT
       mul   $v0, $s0, $v0
       lw    $a0, 0($sp)
       lw    $ra, 4($sp)
       addi  $sp, $sp, 8
       jr    $ra
```

at label HERE, after calling function FACT with input of 4:

old $sp =>	0xnnnnnnnn	???
	– 4	contents of register $ra
$sp =>	– 8	contents of register $a0

at label HERE, after calling function FACT with input of 3:

old $sp =>	0xnnnnnnnn	???
	– 4	contents of register $ra
	– 8	contents of register $a0
	– 12	contents of register $ra
$sp =>	– 16	contents of register $a0

at label HERE, after calling function FACT with input of 2:

old $sp =>	0xnnnnnnnn	???
	– 4	contents of register $ra
	– 8	contents of register $a0
	– 12	contents of register $ra
	– 16	contents of register $a0
	– 20	contents of register $ra
$sp =>	– 24	contents of register $a0

at label HERE, after calling function FACT with input of 1:

old $sp =>	0xnnnnnnnn	???
	– 4	contents of register $ra
	– 8	contents of register $a0
	– 12	contents of register $ra
	– 16	contents of register $a0
	– 20	contents of register $ra
	– 24	contents of register $a0
	– 28	contents of register $ra
$sp =>	– 32	contents of register $a0

2.20.4

| a. | ```
FIB:   addi $sp, $sp,      – 12
       sw    $ra, 8($sp)
       sw    $s1, 4($sp)
       sw    $a0, 0($sp)

       slti $t0, $a0, 3
       beq  $t0, $0, L1
       addi $v0, $0, 1
       j    EXIT

L1:    addi $a0, $a0,      – 1
       jal  FIB
       addi $s1, $v0, $0
       addi $a0, $a0,      – 1

       jal  FIB
       add  $v0, $v0, $s1

EXIT:  lw   $a0, 0($sp)
       lw   $s1, 4($sp)
       lw   $ra, 8($sp)
       addi $sp, $sp, 12
       jr   $ra
``` |
| b. | ```
FIB:   addi $sp, $sp,      – 12
       sw    $ra, 8($sp)
       sw    $s1, 4($sp)
       sw    $a0, 0($sp)

       slti $t0, $a0, 3
       beq  $t0, $0, L1
       addi $v0, $0, 1
       j    EXIT

L1:    addi $a0, $a0,      – 1
       jal  FIB
       addi $s1, $v0, $0
       addi $a0, $a0,      – 1

       jal  FIB
       add  $v0, $v0, $s1

EXIT:  lw   $a0, 0($sp)
       lw   $s1, 4($sp)
       lw   $ra, 8($sp)
       addi $sp, $sp, 12
       jr   $ra
``` |

2.20.5

<table>
<tr>
<td>a.</td>
<td>

23 MIPS instructions to execute nonrecursive vs. 73 instructions to execute (corrected version of) recursion

Nonrecursive version:

```
FIB:   addi  $sp, $sp,         – 4
       sw    $ra, ($sp)
       addi  $s1, $0, 1
       addi  $s2, $0, 1
LOOP:  slti  $t0, $a0, 3
       bne   $t0, $0, EXIT
       add   $s3, $s1, $0
       add   $s1, $s1, $s2
       add   $s2, $s3, $0
       addi  $a0, $a0,         – 1
       j LOOP
EXIT:  add   $v0, s1, $0
       lw    $ra, ($sp)
       addi  $sp, $sp, 4
       jr    $ra
```

</td>
</tr>
<tr>
<td>b.</td>
<td>

23 MIPS instructions to execute nonrecursive vs. 73 instructions to execute (corrected version of) recursion

Nonrecursive version:

```
FIB:   addi  $sp, $sp,         – 4
       sw    $ra, ($sp)
       addi  $s1, $0, 1
       addi  $s2, $0, 1
LOOP:  slti  $t0, $a0, 3
       bne   $t0, $0, EXIT
       add   $s3, $s1, $0
       add   $s1, $s1, $s2
       add   $s2, $s3, $0
       addi  $a0, $a0,         – 1
       j LOOP
EXIT:  add   $v0, s1, $0
       lw    $ra, ($sp)
       addi  $sp, $sp, 4
       jr    $ra
```

</td>
</tr>
</table>

2.20.6

a.	recursive version
	
	``
	FIB: addi $sp, $sp, – 12
	       sw $ra, 8($sp)
	       sw $s1, 4($sp)
	       sw $a0, 0($sp)
	
	HERE: slti $t0, $a0, 3
	       beq $t0, $0, L1
	       addi $v0, $0, 1
	       j EXIT
	
	L1: addi $a0, $a0, – 1
	       jal FIB
	       addi $s1, $v0, $0
	       addi $a0, $a0, – 1
	
	       jal FIB
	       add $v0, $v0, $s1
	
	EXIT: lw $a0, 0($sp)
	       lw $s1, 4($sp)
	       lw $ra, 8($sp)
	       addi $sp, $sp, 12
	       jr $ra
	``
	
	at label HERE, after calling function FIB with input of 4:
	old $sp => 0xnnnnnnnn ???
	                  – 4 contents of register $ra
	                  – 8 contents of register $s1
	$sp => – 12 contents of register $a0

b.	recursive version
	
	``
	FIB: addi $sp, $sp, – 12
	       sw $ra, 8($sp)
	       sw $s1, 4($sp)
	       sw $a0, 0($sp)
	
	HERE: slti $t0, $a0, 3
	       beq $t0, $0, L1
	       addi $v0, $0, 1
	       j EXIT
	
	L1: addi $a0, $a0, – 1
	       jal FIB
	       addi $s1, $v0, $0
	       addi $a0, $a0, – 1
	
	       jal FIB
	       add $v0, $v0, $s1
	
	EXIT: lw $a0, 0($sp)
	       lw $s1, 4($sp)
	       lw $ra, 8($sp)
	       addi $sp, $sp, 12
	       jr $ra
	``
	
	at label HERE, after calling function FIB with input of 4:
	old $sp => 0xnnnnnnnn ???
	                  – 4 contents of register $ra
	                  – 8 contents of register $s1
	$sp => – 12 contents of register $a0

Solution 2.21

2.21.1

a.	after entering function main: old $sp => 0x7ffffffc ??? $sp => – 4 contents of register $ra after entering function leaf_function: old $sp => 0x7ffffffc ??? – 4 contents of register $ra $sp => – 8 contents of register $ra (return to main)
b.	after entering function main: old $sp => 0x7ffffffc ??? $sp => – 4 contents of register $ra after entering function my_function: old $sp => 0x7ffffffc ??? – 4 contents of register $ra $sp => – 8 contents of register $ra (return to main) global pointers: 0x10008000 100 my_global

2.21.2

| a. | ```
MAIN:  addi $sp, $sp,      – 4
       sw   $ra, ($sp)

       addi $a0, $0, 1
       jal  LEAF

       lw   $ra, ($sp)
       addi $sp, $sp, 4
       jr   $ra

LEAF:  addi $sp, $sp,      – 8
       sw   $ra, 4($sp)
       sw   $s0, 0($sp)

       addi $s0, $a0, 1
       slti $t2, 5, $a0
       bne  $t2, $0, DONE
       add  $a0, $s0, $0
       jal  LEAF

DONE:  add  $v0, $s0, $0
       lw   $s0, 0($sp)
       lw   $ra, 4($sp)
       addi $sp, $sp, 8
       jr   $ra
``` |

<table>
<tr><td>b.</td><td>
MAIN: addi $sp, $sp,   – 4
    sw $ra, ($sp)

    addi $a0, $0, 10
    addi $t1, $0, 20
    lw $a1, ($s0)   #assume $s0 has global variable base
    jal FUNC
    add $t2, $v0 $0

    lw $ra, ($sp)
    addi $sp, $sp, 4
    jr $ra

FUNC: sub $v0, $a0, $a1
    jr $ra
</td></tr>
</table>

2.21.3

<table>
<tr><td>a.</td><td>
MAIN: addi $sp, $sp,   – 4
    sw $ra, ($sp)

    addi $a0, $0, 1
    jal LEAF

    lw $ra, ($sp)
    addi $sp, $sp, 4
    jr $ra

LEAF: addi $sp, $sp,   – 8
    sw $ra, 4($sp)
    sw $s0, 0($sp)

    addi $s0, $a0, 1
    slti $t2, 5, $a0
    bne $t2, $0, DONE
    add $a0, $s0, $0
    jal LEAF

DONE: add $v0, $s0, $0
    lw $s0, 0($sp)
    lw $ra, 4($sp)
    addi $sp, $sp, 8
    jr $ra
</td></tr>
<tr><td>b.</td><td>
MAIN: addi $sp, $sp,   – 4
    sw $ra, ($sp)

    addi $a0, $0, 10
    addi $t1, $0, 20
    lw $a1, ($s0)   #assume $s0 has global variable base
    jal FUNC
    add $t2, $v0 $0

    lw $ra, ($sp)
    addi $sp, $sp, 4
    jr $ra

FUNC: sub $v0, $a0, $a1
    jr $ra
</td></tr>
</table>

2.21.4

a.	Register $s0 is used to hold a temporary result without saving $s0 ? rst. To correct this problem, $t0 (or $v0) should be used in place of $s0 in the ? rst two instructions. Note that a sub-optimal solution would be to continue using $s0, but add code to save/restore it.
b.	The two addi instructions move the stack pointer in the wrong direction. Note that the MIPS calling convention requires the stack to grow down. Even if the stack grew up, this code would be incorrect because $ra and $s0 are saved according to the stack-grows-down convention.

2.21.5

a.	int f(int a, int b, int c, int d){
	    return 2*(a – d)+c – b;
	}

b.	int f(int a, int b, int c){
	    return g(a,b)+c;
	}

2.21.6

a.	The function returns 842 (which is 2 × (1 – 30) + 1000 – 100)
b.	The function returns 1500 (g(a, b) is 500, so it returns 500 + 1000)

Solution 2.22

2.22.1

a.	65 20 98 121 116 101
b.	99 111 109 112 117 116 101 114

2.22.2

a.	U+0041, U+0020, U+0062, U+0079, U+0074, U+0065
b.	U+0063, U+006f, U+006d, U+0070, U+0075, U+0074, U+0065, U+0072

2.22.3

a.	add
b.	shift

Solution 2.23

2.23.1

<table>
<tr>
<td>a.</td>
<td>

```
MAIN:  addi $sp, $sp,    – 4
       sw  $ra, ($sp)
       add $t6, $0, 0x30 # '0'
       add $t7, $0, 0x39 # '9'
       add $s0, $0, $0
       add $t0, $a0, $0

LOOP:  lb  $t1, ($t0)
       slt $t2, $t1, $t6
       bne $t2, $0, DONE
       slt $t2, $t7, $t1
       bne $t2, $0, DONE
       sub $t1, $t1, $t6

       beq $s0, $0, FIRST
       mul $s0, $s0, 10
FIRST: add  $s0, $s0, $t1
       addi $t0, $t0, 1
       j LOOP

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

</td>
</tr>
<tr>
<td>b.</td>
<td>

```
MAIN:  addi $sp, $sp,    – 4
       sw  $ra, ($sp)
       add $t4, $0, 0x41 # 'A'
       add $t5, $0, 0x46 # 'F'
       add $t6, $0, 0x30 # '0'
       add $t7, $0, 0x39 # '9'
       add $s0, $0, $0
       add $t0, $a0, $0

LOOP:  lb  $t1, ($t0)
       slt $t2, $t1, $t6
       bne $t2, $0, DONE
       slt $t2, $t7, $t1
       bne $t2, $0, HEX
       sub $t1, $t1, $t6
       j DEC
HEX:   slt $t2, $t1, $t4
       bne $t2, $0, DONE
       slt $t2, $t5, $t1
       bne $t2, $0, DONE
       sub $t1, $t1, $t4
       addi $t1, $t1, 10
DEC:   beq $s0, $0, FIRST
       mul $s0, $s0, 10
FIRST: add  $s0, $s0, $t1
       addi $t0, $t0, 1
       j LOOP

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

</td>
</tr>
</table>

Solution 2.24

2.24.1

a.	0x00000012
b.	0x12ffffff

2.24.2

a.	0x00000080
b.	0x80000000

2.24.3

a.	0x00000011
b.	0x11555555

Solution 2.25

2.25.1   Generally, all solutions are similar:

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

2.25.2   Jump can go up to 0x0FFFFFFC.

a.	no
b.	no

2.25.3   Range is 0x604+ 0x1FFFC = 0x0002 0600 to 0x604 - 0x20000 = 0xFFFE 0604.

a.	no
b.	yes

2.25.4   Range is 0x0042 0600 to 0x003E 0600.

a.	no
b.	no

2.25.5   Generally, all solutions are similar:

```
add  $t1, $zero, $zero     #clear $t1
addi $t2, $zero, top_8_bits  #set top 8b
sll  $t2, $t2, 24          #shift left 24 spots
or   $t1, $t1, $t2         #place top 8b into $t1
addi $t2, $zero, nxt1_8_bits #set next 8b
sll  $t2, $t2, 16          #shift left 16 spots
or   $t1, $t1, $t2         #place next 8b into $t1
addi $t2, $zero, nxt2_8_bits #set next 8b
sll  $t2, $t2, 24          #shift left 8 spots
or   $t1, $t1, $t2         #place next 8b into $t1
ori  $t1, $t1, bot_8_bits    #or in bottom 8b
```

2.25.6

a.	0x12345678
b.	0x12340000

2.25.7

a.	t0 = (0x1234 << 16) || 0x5678;
b.	t0 = (t0 || 0x5678); t0 = 0x1234 << 16;

Solution 2.26

2.26.1   Branch range is 0x00020000 to 0xFFFE0004.

a.	one branch
b.	three branches

2.26.2

a.	one
b.	can't be done

2.26.3   Branch range is 0x00000200 to 0xFFFFFE04.

a.	eight branches
b.	512 branches

2.26.4

a.	branch range is 16x larger
b.	branch range is 16x smaller

2.26.5

a.	no change
b.	jump to addresses 0 to 2^{12} instead of 0 to 2^{28}, assuming the PC<0x08000000

2.26.6

a.	rs ?eld now 3 bits
b.	no change

Solution 2.27

2.27.1

a.	jump register
b.	beq

2.27.2

a.	R-type
b.	I-type

2.27.3

a.	+ can jump to any 32b address – need to load a register with a 32b address, which could take multiple cycles
b.	+ allows the PC to be set to the current PC + 4 +/– BranchAddr, supporting quick forward and backward branches – range of branches is smaller than large programs

2.27.4

a.	0x00000000 lui $s0, 100 0x00000004 ori $s0, $s0, 40	0x3c100100 0x36100028
b.	0x00000100 addi $t0, $0, 0x0000 0x00000104 lw $t1, 0x4000($t0)	0x20080000 0x8d094000

2.27.5

a.	addi $s0, $zero, 0x80 sll $s0, $s0, 17 ori $s0, $s0, 40

b.	addi $t0, $0, 0x0040
	sll $t0, $t0, 8
	lw $t1, 0($t0)

2.27.6

a.	1
b.	1

Solution 2.28

2.28.1

a.	4 instructions

2.28.2

a.	One of the locations speci? ed by the LL instruction has no corresponding SC instruction.

2.28.3

a.	try: MOV R3,R4
	        MOV R6,R7
	        LL R2,0(R2)
	        # adjustment or test code here
	        SC R3,0(R2)
	        BEQZ R3,try
	    try2:
	        LL R5,0(R1)
	        # adjustment or test code here
	        SC R6,0(R1)
	        BEQZ R6,try2
	        MOV R4,R2
	        MOV R7,R5

2.28.4

a.

Processor 1	Processor 2	Cycle	Processor 1		Mem	Processor 2	
			$t1	$t0	($s1)	$t1	$t0
		0	1	2	99	30	40
ll $t1, 0($s1)	ll $t1, 0($s1)	1	99	2	99	99	40
sc $t0, 0($s1)		2	99	1	2	99	40
	sc $t0, 0($s1)	3	99	1	2	99	0

b.

Processor 1	Processor 2	Cycle	Processor 1			Mem	Processor 2		
			$s4	$t1	$t0	($s1)	$s4	$t1	$t0
		0	2	3	4	99	10	20	30
	try: add $t0, $0, $s4	1	2	3	4	99	10	20	10
try: add $t0, $0, $s4	ll $t1, 0($s1)	2	2	3	2	99	10	99	10
ll $t1, 0($s1)		3	2	99	2	99	10	99	10
sc $t0, 0($s1)		4	2	99	1	2	10	99	10
beqz $t0, try	sc $t0, 0($s1)	5	2	99	1	2	10	99	0
add $s4, $0, $t1	beqz $t0, try	6	99	99	1	2	10	99	0

Solution 2.29

2.29.1   The critical section can be implemented as:

```
trylk: li   $t1,1
       ll   $t0,0($a0)
       bnez $t0,trylk
       sc   $t1,0($a0)
       beqz $t1,trylk

       operation

       sw   $zero,0($a0)
```

Where operation is implemented as:

a.	lw $t0,0($a1)
	add $t0,$t0,$a2
	sw $t0,0($a1)

b.	lw $t0,0($a1)
	sge $t1,$t0,$a2
	bnez $t1,skip
	sw $a2,0($a1)
	skip:

2.29.2   The entire critical section is now:

a.	`try:  ll  $t0,0($a1)`
	`      add  $t0,$t0,$a2`
	`      sc   $t0,0($a1)`
	`      beqz $t0,try`

b.	`try:  ll  $t0,0($a1)`
	`      sge  $t1,$t0,$a2`
	`      bnez $t1,skip`
	`      mov  $t0,$a2`
	`      sc   $t0,0($a1)`
	`      beqz $t0,try`
	`skip:`

2.29.3   The code that directly uses ll/sc **to update** shvar **avoids the entire lock/ unlock code. When SC is executed, this code needs 1) one extra instruction to check the outcome of SC, and 2) if the register used for SC is needed again we need an instruction to copy its value. However, these two additional instructions may not be needed, e.g., if SC is not on the best-case path or f it uses a register whose value is no longer needed. We have:**

	Lock-based	Direct LL/SC implementation
a.	6+3	4
b.	6+3	3

2.29.4

a.	Both processors attempt to execute SC at the same time, but one of them completes the write ?rst. The other 's SC detects this and its SC operation fails.
b.	It is possible for one or both processors to complete this code without ever reaching the SC instruction. If only one executes SC, it completes successfully. If both reach SC, they do so in the same cycle, but one SC completes ? rst and then the other detects this and fails.

2.29.5   Every processor has a different set of registers, so a value in a register can-not be shared. Therefore, shared variable shvar **must be kept in memory, loaded each time their value is needed, and stored each time a task wants to change the value of a shared variable. For local variable x there is no such restriction. On the contrary, we want to minimize the time spent in the critical section (or between the LL and SC, so if variable x is in memory it should be loaded to a register before the critical section to avoid loading it during the critical section.**

2.29.6   If we simply do two instances of the code from 2.29.2 one after the other (to update one shared variable and then the other), each update is performed atomically, but the entire two-variable update is not atomic, i.e., after the update to the ? rst variable and before the update to the second variable, another process can perform its own update of one or both variables. If we attempt to do two LLs

(one for each variable), compute their new values, and then do two SC instructions (again, one for each variable), the second LL causes the SC that corresponds to the ?rst LL to fail (we have a LL and SC with a non-register-register instruction executed between them). As a result, this code can never successfully complete.

Solution 2.30

2.30.1

a.	add $t1, $t2, $0
b.	add $t0, $0, small beq $t1, $t0, LOOP

2.30.2

a.	Yes. The address of v is not known until the data segment is built at link time.
b.	No. The branch displacement does not depend on the placement of the instruction in the text segment.

Solution 2.31

2.31.1

a.

	Text Size	0x440
	Data Size	0x90
Text	Address	Instruction
	0x00400000	lw $a0, 0x8000($gp)
	0x00400004	jal 0x0400140
	…	…
	0x00400140	sw $a1, 0x8040($gp)
	0x00400144	jal 0x0400000
	…	…
Data	0x10000000	(X)
	…	…
	0x10000040	(Y)

b.

	Text Size	0x440
	Data Size	0x90
Text	Address	Instruction
	0x00400000	lui $at, 0x1000
	0x00400004	ori $a0, $at, 0
	0x00400008	jal 0x0400140
	…	…
	0x00400140	sw $a0, 8040($gp)
	0x00400144	jmp 0x04002C0
	…	…
	0x004002C0	jr $ra
	…	…
Data	0x10000000	(X)
	…	…
	0x10000040	(Y)

2.31.2   0x8000 data, 0xFC00000 text. However, because of the size of the   beq immediate ? eld, 218 words is a more practical program limitation.

2.31.3   The limitation on the sizes of the displacement and address ?   elds in the instruction encoding may make it impossible to use branch and jump instructions for objects that are linked too far apart.

Solution 2.32

2.32.1

a.	swap:
	sll $t0,$a1,2
	add $t0,$t0,$a0
	lw $t2,0($t0)
	sll $t1,$a2,2
	add $t1,$t1,$a0
	lw $t3,0($t1)
	sw $t3,0($t0)
	sw $t2,0($t1)
	jr $ra

b.	swap:
	lw $t0,0($a0)
	lw $t1,4($a0)
	sw $t1,0($a0)
	sw $t0,4($a0)
	jr $ra

2.32.2

a.	Pass j+1 as a third parameter to swap. We can do this by adding an " addi $a2,$a1,1 " instruction right before " jal swap " .
b.	Pass the address of v[j] to swap. Since that address is already in $t2 at the point when we want to call swap, we can replace the two parameter-passing instructions before " jal swap " with a simple " mov $a0,$t2 " .

2.32.3

a.	swap:
	add $t0,$t0,$a0 ; No sll
	lb $t2,0($t0) ; Byte – sized load
	add $t1,$t1,$a0 ; No sll
	lb $t3,0($t1)
	sb $t3,0($t0) ; Byte – sized store
	sb $t2,0($t1)
	jr $ra

b.	swap: lb $t0,0($a0) ; Byte – sized load lb $t1,1($a0) ; Offset is 1, not 4 sb $t1,0($a0) ; Byte – sized store sb $t0,1($a0) jr $ra

2.32.4

a.	Yes, we must save the additional s-registers. Also, the code for sort() in Figure 2.27 is using 5 t-registers and only 4 s-registers remain. Fortunately, we can easily reduce this number, e.g., by using t1 instead of t0 for loop comparisons.
b.	No change to saving/restoring code is needed because the same s-registers are used in the modi?ed sort() code.

2.32.5 When the array is already sorted, the inner loop always exits in its ?rst iteration, as soon as it compares v[j] with v[j+1]. We have:

a.	We need 4 more instructions to save and 4 more to restore registers. The number of instructions in the rest of the code is the same, so there are exactly 8 more instructions executed in the modi?ed sort(), regardless of how large the array is.
b.	One fewer instruction is executed in each iteration of the inner loop. Because the array is already sorted, the inner loop always exits during its ?rst iteration, so we save one instruction per iteration of the outer loop. Overall, we execute 10 instructions fewer.

2.32.6 When the array is sorted in reverse order, the inner loop always executes the maximum number of iterations and swap is called in each iteration of the inner loop (a total of 45 times). We have:

a.	This change only affects the number of instructions needed to save/restore registers in swap(), so the answer is the same as in Problem When the array is already sorted, the inner loop always exits in its ?rst iteration, as soon as it compares v[j] with v[j+1]. We have:.

b.	One fewer instruction is executed each time the " j>=0" condition for the inner loop is checked. This condition is checked a total of 55 times (whenever swap is called, plus a total of 10 times to exit the inner loop once in each iteration of the outer loop), so we execute 55 instructions fewer.

Solution 2.33

2.33.1

| a. | ```
?nd:  move $v0,$zero
loop: beq  $v0,$a1,done
      sll  $t0,$v0,2
      add  $t0,$t0,$a0
      lw   $t0,0($t0)
      bne  $t0,$a2,skip
      jr   $ra
skip: addi $v0,$v0,1
      b    loop
done: li   $v0,    – 1
      jr   $ra
``` |
| --- | --- |
| b. | ```
count: move $v0,$zero
       move $t0,$zero
loop:  beq  $t0,$a1,done
       sll  $t1,$t0,2
       add  $t1,$t1,$a0
       lw   $t1,0($t1)
       bne  $t1,$a2,skip
       addi $v0,$v0,1
skip:  addi $t0,$t0,1
       b    loop
done:  jr   $ra
``` |

2.33.2

| a. | ```
int ?    nd(int *a, int n, int x){
  int *p;
  for(p=a;p!=a+n;p++)
    if(*p=    =x)
      return p    – a;
  return    – 1;
}
``` |
| --- | --- |
| b. | ```
int count(int *a, int n, int x){
  int res=0;
  int *p;
  for(p=a;p!=a+n;p++)
    if(*p=    =x)
      res=res+1;
  return res;
}
``` |

2.33.3

| a. | ```
?nd:  move $t0,$a0
      sll  $t1,$a1,2
      add  $t1,$t1,$a0
loop: beq  $t0,$t1,done
      lw   $t2,0($t0)
      bne  $t2,$a2,skip
      sub  $v0,$t0,$a0
      srl  $v0,$v0,2
      jr   $ra
skip: addi $t0,$t0,4
      b loop
done: li   $v0,      – 1
      jr   $ra
``` |
| b. | ```
?nd:  move $v0,$zero
      move $t0,$a0
      sll  $t1,$a1,2
      add  $t1,$t1,$a0
loop: beq  $t0,$t1,done
      lw   $t2,0($t0)
      bne  $t2,$a2,skip
      addi $v0,$v0,1
skip: addi $t0,$t0,4
      b    loop
done: jr   $ra
``` |

2.33.4

	Array-based	Pointer-based
a.	7	5
b.	8	6

2.33.5

	Array-based	Pointer-based
a.	1	3
b.	2	3

2.33.6 Nothing would change. The code would change to save all t-registers we use to the stack, but this change is outside the loop body. The loop body itself would stay exactly the same.

Solution 2.34

2.34.1

a.	addi $s0, $0, 10 LOOP: add $s0, $s0, $s1 addi $s0, $s0, – 1 bne $s0, $0, LOOP
b.	sll $s1, $s2, 28 srl $s2, $s2, 4 or $s1, $s1, $s2

2.34.2

a.	ADD, SUBS, MOV—all ARM register-register instruction format BNE—an ARM branch instruction format
b.	ROR—an ARM register-register instruction format

2.34.3

a.	CMP r0, r1 BMI FARAWAY
b.	ADD r0, r1, r2

2.34.4

a.	CMP—an ARM register-register instruction format BMI—an ARM branch instruction format
b.	ADD—an ARM register-register instruction format

Solution 2.35

2.35.1

a.	register operand
b.	register + offset and update register

2.35.2

a.	lw $s0, ($s1)

b.	lw $s1, ($s0)
	lw $s2, 4($s0)
	lw $s3, 8($s0)

2.35.3

a.	addi $s0, $0, TABLE1
	addi $s1, $0, 100
	xor $s2, $s2, $s2
	ADDLP: lw $s4, ($s0)
	addi $s2, $s2, 4
	addi $s0, $s0, 4
	addi $s1, $s1, – 1
	bne $s1, $0, ADDLP

b.	sll $s1, $s2, 28 srl $s2, $s2, 4 or $s1, $s1, $s2

2.35.4

a.	8 ARM vs. 8 MIPS instructions
b.	1 ARM vs. 3 MIPS instructions

2.35.5

a.	ARM 0.67 times as fast as MIPs
b.	ARM 2 times as fast as MIPs

Solution 2.36

2.36.1

a.	sll $s1, $s1, 3 add $s3, $s2, $s1
b.	sll $s4, $s1, 29 srl $s1, $s1, 3 or $s1, $s1, $s4 add $s3, $s2, $s1

2.36.2

a.	addi $s3, $s2, 64
b.	addi $s3, $s2, 64

2.36.3

a.	sll $s1, $s1, 3 add $s3, $s2, $s1
b.	sll $s4, $s1, 29 srl $s1, $s1, 3 or $s1, $s1, $s4 add $s3, $s2, $s1

2.36.4

a.	add r3, r2, #1
b.	add r3, r2, 0x8000

Solution 2.37

2.37.1

a.	mov edx, [esi+4*ebx]	edx=memory(esi+4*ebx)

b.	START: mov ax, 00101100b
	mov cx, 00000011b
	mov bx, 11110000b
	and ax, bx
	or ax, cx
		char ax = 00101100b;
		char bx = 11110000b;
		char cx = 00000011b;
		ax = ax && bx;
		ax = ax || cx;

2.37.2

a.	sll $s2, $s2, 2
	add $s4, $s4, $s2
	lw $s3, ($s4)

b.	START: addi $s0, $0, 0x2c
	addi $s2, $0, 0x03
	addi $s1, $0, 0xf0
	and $s0, $s0, $s1
	or $s0, $s0, $s2

2.37.3

a.	mov edx, [esi+4*ebx]	6, 1, 1, 8, 8
b.	add eax, 0x12345678	4, 4, 1, 32

2.37.4

a.	addi $t0, $0, 2
	sll $a0, $a0, $t0
	add $a0, $a0, $a1
	lw $v0, 0($a0)

b.	lui $a0, 0x1234 ori $a0, 0x5678

Solution 2.38

2.38.1

a.	This instruction copies ECX bytes from an array pointed to by ESI to an array pointer by EDI. An example C library function that can easily be implemented using this instruction is memcpy.
b.	This instruction copies ECX elements, where each element is 4 bytes in size, from an array pointed to by ESI to an array pointer by EDI.

2.38.2

a.	loop: lb $t0,0($a2)
	       sb $t0,0($a1)
	       addi $a0,$a0, – 1
	       addi $a1,$a1,1
	       addi $a2,$a2,1
	       bnez $a0,loop

b.	loop: lw $t0,0($a2)
	       sw $t0,0($a1)
	       addi $a0,$a0, – 1
	       addi $a1,$a1,4
	       addi $a2,$a2,4
	       bnez $a0,loop

2.38.3

	x86	MIPS Speed-up	
a.	5	6	1.2
b.	5	6	1.2

2.38.4

	MIPS code	Code size comparison
a.	f: add $v0,$a0,$a1 jr $ra	MIPS: 2 $\times$ 4 = 8 bytes x86: 11 bytes

b.	f: lw $t0,0($a0)
	   lw $t1,0($a1)
	   add $t0,$t0,$t1
	   sw $t0,0($a0)
	   sw $t0,0($a1)
	   jr $ra
		MIPS: 6 $\times$ 4 = 24 bytes
		x86: 19 bytes

2.38.5   In MIPS, we fetch the next two consecutive instructions by reading the next 8 bytes from the instruction memory. In x86, we only know where the   second instruction begins after we have read and decoded the ?rst one, so it is more dif?cult to design a processor that executes multiple instructions in parallel.

2.38.6   Under these assumptions, using x86 leads to a signi?cant slowdown (the speed-up is well below 1):

	MIPS Cycles	x86 Cycles	Speed-up
a.	2	11	0.18
b.	6	19	0.32

Solution 2.39

2.39.1

a.	0.86 seconds
b.	0.78 seconds

2.39.2   Answer is no in all cases. Slows down the computer.

CCT = clock cycle time
ICa = instruction count (arithmetic)
ICls = instruction count (load/store)
ICb = instruction count (branch)

new CPU time = 0.75   old ICa   CPIa   1.1   oldCCT
            + oldICls   CPIls   1.1   oldCCT
            + oldICb   CPIb   1.1   oldCCT

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

2.39.3

a.	113.16%	121.13%
b.	106.85%	110.64%

2.39.4

a.	3
b.	2.65

2.39.5

a.	0.6
b.	1.07

2.39.6

a.	0.2
b.	0.716666667

Solution 2.40

2.40.1

a.	In the ? rst iteration $0 is 0 and the lw fetches a[0]. After that $t0 is 1, the lw uses a non-aligned address triggers a bus error.
b.	In the ? rst iteration $t0 and $t1 point to a[0] b[0], so the lw and sw instructions access a[0], b[0], and then a[0] as intended. In the second iteration $t0 and $t1 point to the next byte in a[0] and b[1], respectively, instead of pointing to a[1] and b[1]. Thus the ? rst lw uses a non-aligned address and causes a bus error. Note that the computation for $t2 (address of a[n]) does not cause a bus error because that address is not actually used to access memory.

2.40.2

a.	Yes, assuming that x is a sign-extended byte value between − 128 and 127. If x is simply a byte value between 0 and 255, the function procedure only works if neither x nor array a contain values outside the range of 0..127.
b.	Yes.

2.40.3

a.	`f: move $v0,$zero`
	`   move $t0,$zero`
	`L: sll  $t1,$t0,2   ; We must multiply the index by 4 before we`
	`   add  $t1,$t1,$a0  ; add it to a[] to form the address for lw`
	`   lw   $t1,0($t1)`
	`   bne  $t1,$a2,S`
	`   addi $v0,$v0,1`
	`S: addi $t0,$t0,1`
	`   bne  $t0,$a1,L`
	`   jr   $ra`

b.	`f: move $t0,$a0`
	`   move $t1,$a1`
	`   sll  $t2,$a2,2     ; We must multiply n by 4 to get the address`
	`   add  $t2,$t2,$a0   ; of the end of array a`
	`L: lw   $t3,0($t0)`
	`   lw   $t4,0($t1)`
	`   add  $t3,$t3,$t4`
	`   sw   $t3,0($t0)`
	`   addi $t0,$t0,4    ; Move to next element in a`
	`   addi $t1,$t1,4    ; Move to next element in b`
	`   bne  $t0,$t2,L`
	`   jr   $ra`

2.40.4 At the exit from my_alloc, the $sp register is moved to " free " the memory that is returned to main. Then my_init() writes to this memory to initialize it. Note that neither my_init nor main access the stack memory in any other way until sort() is called, so the values at the point where sort() is called are still the same as those written by my_init:

a.	0, 0, 0, 0, 0
b.	5, 4, 3, 2, 1

2.40.5   In main, register $s0 becomes 5, then my_alloc   is called. The address of the array v " allocated my_alloc   is 0xffe8, because in my_alloc $sp was saved at 0xfffc, and then 20 bytes (4   5) were reserved for array arr ($sp was dec-remented by 20 to yield 0xffe8). The elements of array v returned to main are thus a[0] at 0xffe8, a[1] at 0xffec, a[2] at 0xfff0, a[3] at 0xfff4, and a[4] at 0xfff8. After my_alloc   returns, $sp is back to 0x10000. The value returned from my_alloc is 0xffe8 and this address is placed into the $s1 register. The my_init   function does not modify $sp, $s0, $s1, $s2, or $s3. When sort()   begins to execute, $sp is 0x1000, $s0 is 5, $s1 is 0xffe7, and $s2 and $s3 keep their original values of 10 and 1, respectively. The sort (0 procedure then changes $sp to 0xffec (0x1000 minus 20), and writes $s0 to memory at address 0xffec (this is where a[1] is, so a[1] becomes 5), writes $s1 to memory at address 0xfff0 (this is where a[2] is, so a[2] becomes 0xffe8), writes $s2 to memory address 0xfff4 (this is where a[3] is, so a[3] becomes - 10), writes $s3 to memory address 0xfff8 (this is where a[4] is, so a[4] becomes 1), and writes the return address to 0xfffc, which does not affect values in array v. Now the values of array v are:

a.	0 5 0xffe8 7 1
b.	5 5 0xffe8 7 1

2.40.6   When the sort()   procedure enters its main loop, the elements of array v are sorted without any interference from other stack accesses. The resulting sorted array is

a.	0, 1, 5, 7, 0xffe8
b.	1, 5, 5, 7, 0xffe8

Unfortunately, this is not the end of the chaos caused by the original bug in my_alloc   . When the sort()   function begins restoring registers, $ra is read the (luckily) unmodi? ed location where it was saved. Then $s0 is read from memory at address 0xffec (this is where a[1] is), $s1 is read from address 0xfff0 (this is where a[2] is), $s2 is read from address 0xfff4 (this is where a[3] is), and $s3 is read from address 0xfff8 (this is where a[4] is). When   sort()   returns to main()   , registers $s0 and $s1 are supposed to keep n and the address of array   v. As a result, after sort()   returns to main()   , n and v are:

a.	n=1, v=5 So v is a 1-element array of integers that begins at address 5
b.	n=5, v=5 So v is a 5-element array of integers that begins at address 5

If we were to actually attempt to access (e.g., print out) elements of array   v in the main()   function after this point, the ?   rst lw would result in a bus error due to non-aligned address. If MIPS were to tolerate non-aligned accesses, we would print out whatever values were at the address v points to (note that this is not the same address to which my_init   wrote its values).

Computer Organization and Design

5th Edition

3

Solutions

3.1   5730

3.2   5730

3.3   0101111011010100

The attraction is that each hex digit contains one of 16 diff erent characters
(0 – 9, A – E). Since with 4 binary bits you can represent 16 diff erent patterns,
in hex each digit requires exactly 4 binary bits. And bytes are by defi nition 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   Overfl ow (result 179, which does not fi t 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   6212

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   6×2×12

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

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

For a soft ware implementation, it takes 1 cycle to decide what to add, 1 cyc
to do the add, 1 cycle to do each shift , and 1 cycle to decide if we are done.
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 soft ware

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

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

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

(Could also use 0x55, which is 64+16+4+1, and shift 0x33 left 6 times, add
to it 0x33 shift ed left 4 times, add to that 0x33 shift ed left 2 times, and ad
that 0x33. Same number of shift s 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) aft er 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

  $0\times$ 0C000000 = 0000 1100 0000 0000 0000 0000 0000 0000

  = 0 0001 1000 0000 0000 0000 0000 0000 000

  sign is positive

  exp = 0 $\times$ 18 = 24 127 = 103

  there is a hidden 1

  mantissa = 0

  answer = 1.0×2^{103}

3.23 63.25×10^{0} 111111.01 $\times 2^{0}$

    normalize, move binary point 5 to the left

    1.1111101×2^{5}

    sign positive, exp 1275132

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

     0100 0010 0111 1101 0000 0000 0000 0000 0x427D0000

3.24 63.25×10^{0} 111111.01 $\times 2^{0}$

    normalize, move binary point 5 to the left

    1.1111101×2^{5}

    sign positive, exp 102351028

    Final bit pattern:

    0 100 0000 0100 1111 1010 0000 0000 0000 0000 0000 0000 0000 0000 00 0000
     0x404FA00000000000

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

move hex point 2 to the left

$.3F40 \times 16^2$

sign = positive, exp = 642

Final bit pattern: 01000010001111110100000000000000

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

$= .00101 \times 2^0$

move the binary point 2 to the right

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

exponent = -2, fraction = .10100000000000000000000

answer: 11111111111101011000000000000000000000

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

$= .00101 \times 2^0$

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

exponent = $-3 = 315 = 12$, fraction = .0100000000

answer: 1011000100000000

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

$= .00101 \times 2^0$

move the binary point 2 to the right

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

exponent = -2, fraction = .10100000000000000000000000000000

answer: 10110000000000000000000000000101

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

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

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

Shift binary point 6 to the left to align exponents,

                              GR

1.1010001000 00

1.0000011010 10 0111 (Guard 5 1, Round 5 0,
Sticky 5 1)

1.1010100010 10

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

Thus, the value is rounded up.

1.1010100011 2^4 11010.100011 2^0 26.546875   2.6546875 10^1

3.30   8.0546875   1.79931640625 10^1

     8.0546875   1.0000000111 2^3

     1.79931640625 10^1 1.0111000010 2^3

     Exp: 3 3 0, 016 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 × 2^0 = 0100000111001100 (1.0111001100 = 1.44921875)

8.0546875 × .179931640625 = 1.4492931365966796875

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

3.31   8.625 × 10^1 / 4.875 × 10^0

8.625 × 10^1 = 1.0101100100 × 2^6

4.875 = 1.0011100000 × 2^2

Exponent = 62 − 4, = 415 − 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
           --------------------
            .0000000110000000
```

1.000110110001001111 Guard0, Round1, Sticky1: No Round, fi x 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 fi t into the available 10-b fi eld. Answer off by .00480769230

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

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

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

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

shift binary point of smaller left 12 so exponents match

(A)   1.1001100000

(B)   1.0110000000

         10.1111100000 Normalize,

(AB)   1.0111110000 $\times$ 2^{1}

(C)   1.1011101011

(AB)   .0000000000   10 111110000 Guard 1, Round 0, Sticky 1

(AB)C 1.1011101011 10 1 Round up

(AB)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

(BC)   1.1011101011

(A)   .0000000000 011001100000

A(BC) 1.1011101011 No round

A(BC) 1.1011101011 2^{10} 0110101011101011 1771

3.34   No, they are not equal: (AB)C 1772, A(BC) 1771 (steps shown above).

Exact: .398437 .34375 1771 1771.742187

3.35   (3.41796875 10^3 6.34765625 10^{-3}) 1.05625 10^{-2}

(A) 3.41796875 10^{-3} 1.1100000000 2^{-9}

(B) 4.150390625 10^{-3} 1.0001000000 2^{-8}

(C) 1.05625 10^{-2} 1.1010011010 2^{-6}

Exp: 98 17

Signs: both positive, result positive

Fraction:

(A)   1.1100000000

(B)   1.0001000000

               11100000000

            11100000000

            1.11011100000000000000

AB   1.1101110000 00 00000000

Guard 0, Round 0, Sticky 0: No Round

AB 1.1101110000 2^{-17} UNDERFLOW: Cannot represent number

3.36   3.41796875 10^3 (6.34765625 10^{-3} 1.05625 10^{-2})

(A)   3.41796875 10^{-3} 1.1100000000 2^{-9}

(B)   4.150390625 10^{-3} 1.0001000000 2^{-8}

(C)   1.05625 10^{-2} 1.1010011010 2^{-6}

Exp: 86 2

Signs: both positive, result positive

Fraction:
(B)   1.0001000000
(C)   1.1010011010

            10001000000
          10001000000
          10001000000
        10001000000
      10001000000
    10001000000

1.110000001110100000000
 1.1100000011 10 100000000 Guard 5 1, Round 5 0, Sticky
5 1: Round

BC   1.1100000100 2^{2}

Exp: 92 11

Signs: both positive, result positive

Fraction:
(A)   1.1100000000
(B x C) 1.1100000100

            11100000000
        11100000000
        11100000000
      11100000000

11.000100011100000000000   Normalize, add 1 to exponent
 1.1000100011 10 0000000000 Guard=1, Round=0, Sticky=0:
Round to even

 A(BC) 1.1000100100 2^{10}

3.37   b) No:

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

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

A and B are both small, so their product does not fi t into the

16-bit fl oating 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 shift ing necessary
(B)   1.0011010011
(C) $-$1.0011010010

(BC)   0.0000000001 $\times 2^{-14}$
(BC)   1.0000000000 $\times 2^{-4}$

Exp: 04 $-$4

Signs: both positive, result positive

Fraction:
(A)   1.1010101010
(BC)   1.0000000000

      11010101010

              1.10101010100000000000
A(BC)   1.1010101010 0000000000 Guard 0, Round
          0, sticky 0: No round

A(BC) $1.1010101010 \times 2^{-4}$

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

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

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

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

Exp: $014 = 14$

Signs: both positive, result positive

Fraction:

(A)   1.1010101010

(B)   1.0011010011

                 11010101010

                 11010101010

                11010101010

               11010101010

                        11010101010

             11010101010

                 10.0000001001100001111   Normalize, add 1 to
                                                  exponent

AB   1.0000000100 11 00001111   Guard = 1, Round = 1,
                                                     Sticky = 1: Round

AB   1.0000000101 × 2^{15}

Exp: $014 \to 14$

Signs: one negative, one positive, result negative

Fraction:

(A)   1.1010101010

(C)   1.0011010010

                                           11010101010

                    11010101010

                    11010101010

                    11010101010

                                     11010101010

              10.0000000111110111010

Normalize, add 1 to exponent

AC   1.0000000011 11 101110100

Guard = 1, Round = 1, Sticky = 1: Round

AC   1.0000000100 × 2^{15}

AB   1.0000000101 × 2^{15}

AC   1.0000000100 × 2^{15}

ABAC   .0000000001 × 2^{15}

ABAC   1.0000000000 × 2^{5}

3.40   b) No:

A(BC) 1.1010101010 2^4 26.65625, and (AB)(AC)
1.0000000000 2^5 32

Exact: 1.666015625 (19,760 19,744) 26.65625

3.41

Answer	sign	exp	Exact?
1 01111101 00000000000000000000000	–	2	Yes

3.42   bbbb 1

b4 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

Solution 4.1

4.1.1 The values of the signals are as follows:

	RegWrite	MemRead	ALUMux	MemWrite	ALUOp	RegMux	Branch
a.	1	0	0 (Reg)	0	Add	1 (ALU)	0
b.	1	1	1 (Imm)	0	Add	1 (Mem)	0

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

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

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

4.1.2 Resources performing a useful function for this instruction are:

a.	All except Data Memory and branch Add unit
b.	All except branch Add unit and second read port of the Registers

4.1.3

	Outputs that are not used	No outputs
a.	Branch Add	Data Memory
b.	Branch Add, second read port of Registers	None (all units produce outputs)

4.1.4
One long path for and instruction is to read the instruction, read the registers, go through the ALUMux, perform the ALU operation, and go through the Mux that controls the write data for Registers (I-Mem, Regs, Mux, ALU, and Mux). The other long path is similar, but goes through Control while registers are read (I- Mem, Control, Mux, ALU, Mux). There are other paths but they are shorter, such as the PC increment path (only Add and then Mux), the path to prevent branching (I-Mem, Control, Mux uses Branch signal to select the PC + 4 input as the new value for PC), the path that prevents a memory write (only I-Mem and then Control, etc).

a.	Control is faster than registers, so the critical path is I-Mem, Regs, Mux, ALU, Mux.
b.	Control is faster than registers, so the critical path is I-Mem, Regs, Mux, ALU, Mux.

4.1.5   One long path is to read instruction, read registers, use the Mux to select the immediate as the second ALU input, use ALU (compute address), access D-Mem, and use the Mux to select that as register data input, so we have I-Mem, Regs, Mux, ALU, D-Mem, Mux. The other long path is similar, but goes through Control instead of Regs (to generate the control signal for the ALU MUX). Other paths are shorter, and are similar to shorter paths described for 4.1.4.

a.	Control is faster than registers, so the critical path is I-Mem, Regs, Mux, ALU, D-Mem, Mux.
b.	Control is faster than registers, so the critical path is I-Mem, Regs, Mux, ALU, Mux.

4.1.6   This instruction has two kinds of long paths, those that determine the branch condition and those that compute the new PC. To determine the branch condition, we read the instruction, read registers or use the Control unit, then use the ALU Mux and then the ALU to compare the two values, then use the Zero output of the ALU to control the Mux that selects the new PC. As in 4.1.4 and 4.1.5:

a.	The ?rst path (through Regs) is longer.
b.	The ?rst path (through Regs) is longer.

To compute the PC, one path is to increment it by 4 (Add), add the offset (Add), and select that value as the new PC (Mux). The other path for computing the PC is to Read the instruction (to get the offset), use the branch Add unit and Mux. Both of the compute-PC paths are shorter than the critical path that determines the branch condition, because I-Mem is slower than the PC + 4 Add unit, and because ALU is slower than the branch Add.

Solution 4.2

4.2.1   Existing blocks that can be used for this instruction are:

a.	This instruction uses instruction memory, both existing read ports of Registers, the ALU, and the write port of Registers.
b.	This instruction uses the instruction memory, one of the existing register read ports, the path that passed the immediate to the ALU, and the register write port.

4.2.2   New functional blocks needed for this instruction are:

a.	Another read port in Registers (to read Rx) and either a second ALU (to add Rx to Rs + Rt) or a third input to the existing ALU.
b.	We need to extend the existing ALU to also do shifts (adds a SLL ALU operation).

4.2.3   The new control signals are:

a.	We need a control signal that tells the new ALU what to do, or if we extended the existing ALU we need to add a new ADD3 operation.
b.	We need to change the ALU Operation control signals to support the added SLL operation in the ALU.

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

New clock cycle time	
a.	1130ps (No change, Add units are not on the critical path).
b.	1230 (1130ps + 100ps, Regs are on the critical path)

4.2.5   The speed-up comes from changes in clock cycle time and changes to the number of clock cycles we need for the program:

Bene?t	
a.	Speed-up is 1 (no change in number of cycles, no change in clock cycle time).
b.	We need 5% fewer cycles for a program, but cycle time is 1230 instead of 1130, so we have a speed-up of (1/0.95) × (1130/1230) = 0.97, which means we actually have a small slowdown.

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

We will compute cost relative to this baseline. The performance relative to this baseline is the speed-up we computed in 4.2.5, and our cost/performance relative to the baseline is as follows:

	New cost	Relative cost	Cost/Performance
a.	3890 + 2 × 20 = 3930	3930/3890 = 1.01	1.01/1 = 1.01. We are paying a bit more for the same performance.
b.	3890 + 200 = 4090	4090/3890 = 1.05	1.05/0.97 = 1.08. We are paying some more and getting a small slowdown, so out cost/ performance gets worse.

Solution 4.3

4.3.1

a.	Both. It is mostly ? ip-?ops, but it has logic that controls which ? ip-?ops get read or written in each cycle
b.	Both. It is mostly ? ip-?ops, but it has logic that controls which ? ip-?ops get read or written in each cycle

4.3.2

a.

This shows the lowermost bit of each word. This schematic is repeated 7 more times for the remaining seven bits. Note that there are no connections for D and C ? ip-? op inputs because datapath ? gures do not specify how instruction memory is written.

b.

This is the schematic for the lowermost bit, it needs to be repeated 7 more times for the remaining bits. RReg1 is the Read Register 1 input, RReg2 is the Read Register 2 input, WReg is the Write Register input, WData is the Write Data input. RData1 and RData2 are Read Data 1 and Read Data 2 outputs. Data outputs and input have   " _0"   to denote that this is only bit 0 of the 8-bit signal.

4.3.3

a.

A1   A0

Instruction 0

b.   No change, there are no gates with more then 2 inputs in the schematic.

4.3.4   The latency of a path is the latency from an input (or a D-element output) to an output (or D-element input). The latency of the circuit is the latency of the path with the longest latency. Note that there are many correct ways to design the circuit in 4.3.2, and for each solution to 4.3.2 there is a different solution for this problem.

4.3.5   The cost of the implementation is simply the total cost of all its components. Note that there are many correct ways to design the circuit in 4.3.2, and for each solution to 4.3.2 there is a different solution for this problem.

4.3.6

a.	Because multi-input AND and OR gates have the same latency as 2-input ones, we can use many-input gates to reduce the number of gates on the path from inputs to outputs. The schematic shown for 4.3.2 turns out to already be optimal.
b.	A three-input or a four-input gate has a lower latency than a cascade of two 2-input gates. This means that shorter overall latency is achieved by using 3- and 4-input gates rather than cascades of 2-input gates. In our schematic shown for 4.3.2, we should replace the three 2-input AND gates used for Clock, RegWrite, and WReg signals with two 3-input AND gates that directly determine the value of the C input for each D-element.

Solution 4.4

4.4.1 We show the implementation and also determine the latency (in gates) needed for 4.4.2.

	Implementation	Latency in gates
a.		4
b.		4

4.4.2 See answer for 4.4.1 above.

4.4.3

	Implementation
a.	
b.	

4.4.4

a.	There are four OR gates on the critical path, for a total of 136ps
b.	The critical path consists of OR, XOR, OR, and OR, for a total of 510ps

4.4.5

a.	The cost is 2 AND gates and 4 OR gates, for a total cost of 16.
b.	The cost is 1 AND gate, 4 OR gates, and 1 XOR gate, for a total cost of 12.

4.4.6   We already computed the cost of the combined circuit. Now we determine the cost of the separate circuits and the savings.

	Combinend cost	Separate cost	Saved
a.	16	22 (+2 OR gates)	(22 − 16)/22 = 27%
b.	12	14 (+1 AND gate)	(14 − 12)/14 = 14%

Solution 4.5

4.5.1

4.5.2

4.5.3

	Cycle time	Operation time
a.	90ps (OR, AND, D)	32 × 90ps = 2880ps
b.	170ps (NOT, AND, D)	32 × 170ps = 5440ps

4.5.4

	Cycle time	Speed-up
a.	120ps (OR, AND, AND, D)	(32 × 90ps)/(16 × 120ps) = 1.50
b.	90ps (NOT, AND)	(32 × 170ps)/(16 × 90ps) = 3.78

4.5.5

	Circuit 1	Circuit 2
a.	14 (1 AND, 1 OR, 1 XOR, 1 D)	20 (2 AND, 1 OR, 2 XOR, 1 D)
b.	29 (1 NOT, 2 AND, 2 D)	29 (1 NOT, 2 AND, 2 D)

4.5.6

	Cost/Performance for Circuit 1	Cost/Performance for Circuit 2	Circuit 1 versus Circuit 2
a.	14 × 32 × 90 = 40320	20 × 16 × 120 = 38400	Cost/performance of Circuit 2 is better by about 4.7%
b.	29 × 32 × 170 = 157760	29 × 16 × 90 = 41760	Cost/performance of Circuit 2 is better by about 73.5%

Solution 4.6

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

a.	400ps
b.	500ps

4.6.2   The critical path for this instruction is through the instruction memory, Sign-extend and Shift-left-2 to get the offset, Add unit to compute the new PC, and Mux to select that value instead of PC + 4. Note that the path through the other

Add unit is shorter, because the latency of I-Mem is longer that the latency of the Add unit. We have:

a.	400ps + 20ps + 2ps + 100ps + 30ps = 552ps
b.	500ps + 90ps + 20ps + 150ps + 100ps = 860ps

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

a.	400ps + 200ps + 30ps + 120ps + 30ps = 780ps
b.	500ps + 220ps + 100ps + 180ps + 100ps = 1100ps

4.6.4

a.	All instructions except jumps that are not PC-relative (jal, jalr, j, jr)
b.	Loads and stores

4.6.5

a.	None. I-Mem is slower, and all instructions (even NOP) need to read the instruction.
b.	Loads and stores.

4.6.6   Of the two instruction (bne and add), bne has a longer critical path so it determines the clock cycle time. Note that every path for add is shorter or equal to than the corresponding path for bne , so changes in unit latency will not affect this. As a result, we focus on how the unit 's latency affects the critical path of bne.

a.	This unit is not on the critical path, so changes to its latency do not affect the clock cycle time unless the latency of the unit becomes so large to create a new critical path through this unit, the branch add, and the PC Mux. The latency of this path is 230ps and it needs to be above 780ps, so the latency of the Add-4 unit needs to be more 650ps for it to be on the critical path.
b.	This unit is not used by BNE nor by ADD, so it cannot affect the critical path for either instruction.

Solution 4.7

4.7.1   The longest-latency path for ALU operations is through I-Mem, Regs, Mux (to select ALU operand), ALU, and Mux (to select value for register write). Note that the only other path of interest is the PC-increment path through Add (PC + 4)

and Mux, which is much shorter. So for the I-Mem, Regs, Mux, ALU, Mux path we have:

a.	400ps + 200ps + 30ps + 120ps + 30ps = 780ps
b.	500ps + 220ps + 100ps + 180ps + 100ps = 1100ps

4.7.2 The longest-latency path for lw is through I-Mem, Regs, Mux (to select ALU input), ALU, D-Dem, and Mux (to select what is written to register). The only other interesting paths are the PC-increment path (which is much shorter) and the path through Sign-extend unit in address computation instead of through Registers. However, Regs has a longer latency than Sign-extend, so for I-Mem, Regs, Mux, ALU, D-Mem, and Mux path we have:

a.	400ps + 200ps + 30ps + 120ps + 350ps + 30ps = 1130ps
b.	500ps + 220ps + 100ps + 180ps + 1000ps + 100ps = 2100ps

4.7.3 The answer is the same as in 4.7.2 because the lw instruction has the longest critical path. The longest path for sw is shorter by one Mux latency (no write to register), and the longest path for add or bne is shorter by one D-Mem latency.

4.7.4 The data memory is used by lw and sw instructions, so the answer is:

a.	20% + 10% = 30%
b.	35% + 15% = 50%

4.7.5 The sign-extend circuit is actually computing a result in every cycle, but its output is ignored for add and not instructions. The input of the sign-extend circuit is needed for addi (to provide the immediate ALU operand), beq (to provide the PC-relative offset), and lw and sw (to provide the offset used in addressing memory) so the answer is:

a.	15% + 20% + 20% + 10% = 65%
b.	5% + 15% + 35% + 15% = 70%

4.7.6 The clock cycle time is determined by the critical path for the instruction that has the longest critical path. This is the lw instruction, and its critical path goes through I-Mem, Regs, Mux, ALU, D-Mem, and Mux so we have:

a.	I-Mem has the longest latency, so we reduce its latency from 400ps to 360ps, making the clock cycle 40ps shorter. The speed-up achieved by reducing the clock cycle time is then 1130ps/ 1090ps = 1.037
b.	D-Mem has the longest latency, so we reduce its latency from 1000ps to 900ps, making the clock cycle 100ps shorter. The speed-up achieved by reducing the clock cycle time is then 2100ps/2000ps = 1.050

Solution 4.8

4.8.1   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:

a.	Bit 7 of the instruction word is only used as part of an immediate/offset part of the instruction, so one way to test would be to execute ADDI $1, zero, 128 which is supposed to place a value of 128 into $1. If instruction bit 7 is stuck at zero, $1 will be zero because value 128 has all bits at zero except bit 7.
b.	The only instructions that set this signal to 1 are loads. We can test by ?lling the data memory with zeros and executing a load instruction from a non-zero address, e.g., LW $1, 1024(zero). After this instruction, the value in $1 is supposed to be zero. If the MemtoReg signal is stuck at 0, the value in the register will be 1024 (the Mux selects the ALU output (1024) instead of the value from memory).

4.8.2   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:

a.	We can use ADDI $1, zero, 0 which is supposed to put a value of 0 in $1. If Bit 7 of the instruction word is stuck at 1, the immediate operand becomes 128 and $1 becomes 128 instead of 0.
b.	We cannot reliably test for this fault, because all instructions that set the MemtoReg signal to zero also set the ReadMem signal to zero. If one of these instructions is used as a test for MemtoReg stuck-at-1, the value written to the destination register is "random" (whatever noise is there at the data output of Data Memory). This value could be the same as the value already in the register, so if the fault exists the test may not detect it.

4.8.3

a.	It is possible to work around this fault, but it is very dif?cult. We must ?nd all instructions that have zero in this bit of the offset or immediate operand and replace them with a sequence of "safe" instruction. For example, a load with such an offset must be replaced with an instruction that subtracts 128 from the address register, then the load (with the offset larger by 128 to set bit 7 of the offset to 1), then subtract 128 from the address register.
b.	We cannot work around this problem, because it prevents all instructions from storing their result in registers, except for load instructions. Load instructions only move data from memory to registers, so they cannot be used to emulate ALU operations "broken" by the fault.

4.8.4

a.	If MemRead is stuck at 0, data memory is read for every instruction. However, for non-load instructions the value from memory is discarded by the Mux that selects the value to be written to the Register unit. As a result, we cannot design this kind of test for this fault, because the processor still operates correctly (although inef? ciently).
b.	To test for this fault, we need an instruction whose opcode is zero and MemRead is 1. However, instructions with a zero opcode are ALU operations (not loads), so their MemRead is 0. As a result, we cannot design this kind of test for this fault, because the processor operates correctly.

4.8.5

a.	If Jump is stuck-at-1, every instruction updates the PC as if it were a jump instruction. To test for this fault, we can execute an ADDI with a non-zero immediate operand. If the Jump signal is stuck-at-1, the PC after the ADDI executes will not be pointing to the instruction that follows the ADDI.
b.	To test for this fault, we need an instruction whose opcode is zero and Jump is 1. However, the opcode for the jump instruction is non-zero. As a result, we cannot design this kind of test for this fault, because the processor operates correctly.

4.8.6 Each single-instruction test " covers " all faults that, if present, result in different behavior for the test instruction. To test for as many of these faults as possible in a single instruction, we need an instruction that sets as many of these signals to a value that would be changed by a fault. Some signals cannot be tested using this single-instruction method, because the fault on a signal could still result in completely correct execution of all instruction that trigger the fault.

Solution 4.9

4.9.1

	Binary	Hexadecimal
a.	100011 00110 00001 0000000000101000	8CC10028
b.	000101 00001 00010 1111111111111111	1422FFFF

4.9.2

	Read register 1	Actually read?	Read register 2	Actually read?
a.	6 (00110$_b$)	Yes	1 (00001$_b$)	Yes (but not used)
b.	1 (00001$_b$)	Yes	2 (00010$_b$)	Yes

4.9.3

	Read register 1	Register actually written?
a.	1 (00001 $_b$)	Yes
b.	Either 2 (00010 $_b$) of 31 (11111 $_b$) (don't know because RegDst is X)	No

4.9.4

	Control signal 1	Control signal 2
a.	RegDst = 0	MemRead = 1
b.	RegWrite = 0	MemRead = 0

4.9.5 We use I31 through I26 to denote individual bits of Instruction[31:26], which is the input to the Control unit:

a.	RegDst = NOT I31
b.	RegWrite = (NOT I28 AND NOT I27) OR (I31 AND NOT I29)

4.9.6 If possible, we try to reuse some or all of the logic needed for one signal to help us compute the other signal at a lower cost:

a.	RegDst = NOT I31 MemRead = I31 AND NOT I29
b.	MemRead = I31 AND NOT I29 RegWrite = (NOT I28 AND NOT I27) OR MemRead

Solution 4.10

To solve problems in this exercise, it helps to ?rst determine the latencies of different paths inside the processor. Assuming zero latency for the Control unit, the critical path is the path to get the data for a load instruction, so we have I-Mem, Mux, Regs, Mux, ALU, D-Mem and Mux on this path.

4.10.1 The Control unit can begin generating MemWrite only after I-Mem is read. It must ?nish generating this signal before the end of the clock cycle. Note that MemWrite is actually a write-enable signal for D-Mem ?ip-?ops, and the actual write is triggered by the edge of the clock signal, so MemWrite need not

arrive before that time. So the Control unit must generate the MemWrite in one clock cycle, minus the I-Mem access time:

	Critical path	Maximum time to generate MemWrite
a.	400ps + 30ps + 200ps + 30ps + 120ps + 350ps + 30ps = 1160ps	1160ps − 400ps = 760ps
b.	500ps + 100ps + 220ps + 100ps + 180ps + 1000ps + 100ps = 2200ps	2200ps − 500ps = 1700ps

4.10.2   All control signals start to be generated after I-Mem read is complete. The most slack a signal can have is until the end of the cycle, and MemWrite and RegWrite are both needed only at the end of the cycle, so they have the most slack. The time to generate both signals without increasing the critical path is the one computed in 4.10.1.

4.10.3   MemWrite and RegWrite are only needed by the end of the cycle. RegDst, Jump, and MemtoReg are needed one Mux latency before the end of the cycle, so they are more critical than MemWrite and RegWrite. Branch is needed two Mux latencies before the end of the cycle, so it is more critical than these. MemRead is needed one D-Mem plus one Mux latency before the end of the cycle, and D-Mem has more latency than a Mux, so MemRead is more critical than Branch. ALUOp must get to ALU control in time to allow one ALU Ctrl, one ALU, one D-Mem, and one Mux latency before the end of the cycle. This is clearly more critical than MemRead. Finally, ALUSrc must get to the pre-ALU Mux in time, one Mux, one ALU, one D-Mem, and one Mux latency before the end of the cycle. Again, this is more critical than MemRead. Between ALUOp and ALUSrc, ALUOp is more critical than ALUSrc if ALU control has more latency than a Mux. If ALUOp is the most critical, it must be generated one ALU Ctrl latency before the critical-path signals can go through Mux, Regs, and Mux. If the ALUSrc signal is the most critical, it must be generated while the critical path goes through Mux and Regs. We have

	The most critical control signal is	Time to generate it without affecting the clock cycle time
a.	ALUOp (50ps > 30ps)	30ps + 200ps + 30ps − 50ps = 210ps
b.	ALUSrc (100ps > 55ps)	100ps + 220ps = 320ps

For the next three problems, it helps to compute for each signal how much time we have to generate it before it starts affecting the critical path. We already did this for RegDst and RegWrite in 4.10.1, and in 4.10.3 we described how to do it for the remaining control signals. We have:

	RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
a.	730ps	730ps	700ps	380ps	730ps	210ps	760ps	230ps	760ps
b.	1600ps	1600ps	1500ps	600ps	1600ps	365ps	1700ps	320ps	1700ps

The difference between the allowed time and the actual time to generate the signal is called " . Slack For this problem, the allowed time will be the maximum time the signal can take without affecting clock cycle time. If slack is positive, the signal arrives before it is actually needed and it does not affect clock cycle time. If the slack is positive, the signal is late and the clock cycle time must be adjusted. We now compute the clack for each signal:

	RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
a.	10ps	0ps	100ps	− 20ps	30ps	10ps	50ps	30ps	− 40ps
b.	0ps	0ps	100ps	100ps	200ps	− 35ps	200ps	− 80ps	0ps

4.10.4   With this in mind, the clock cycle time is what we computed in 4.10.1, plus the absolute value of the most negative slack. We have:

	Control signal with the most negative slack is	Clock cycle time with ideal Control unit (from 4.10.1)	Actual clock cycle time with these signal latencies
a.	RegWrite (- 40ps)	1160ps	1200ps
b.	ALUSrc (- 80ps)	2200ps	2280ps

4.10.5   It only makes sense to pay to speed-up signals with negative slack, because improvements to signals with positive slack cost us without improving performance. Furthermore, for each signal with negative slack, we need to speed it up only until we eliminate all its negative slack, so we have:

	Signals with negative slack	Per-processor cost to eliminate all negative slack
a.	MemRead (20ps) RegWrite (− 40ps)	60ps at $1/5ps = $12
b.	ALUOp (35ps) ALUSrc (− 80ps)	115ps at $1/5ps = $23

4.10.6   The signal with the most negative slack determines the new clock cycle time. The new clock cycle time increases the slack of all signals until there are is no remaining negative slack. To minimize cost, we can then slow down signals that end up having some (positive) slack. Overall, the cost is minimized by slowing signals down by:

	RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
a.	50ps	40ps	140ps	20ps	70ps	50ps	90ps	70ps	0ps
b.	80ps	80ps	180ps	180ps	280ps	45ps	280ps	0ps	80ps

Solution 4.11

4.11.1

	Sign-extend	Jump's shift-left-2
a.	00000000000000000000000000010000	0001000011000000000001000000
b.	00000000000000000000000000001100	0000100011000000000000110000

4.11.2

	ALUOp[1-0]	Instruction[5-0]
a.	00 010000	
b.	01	001100

4.11.3

	New PC	Path
a.	PC + 4	PC to Add (PC + 4) to branch Mux to jump Mux to PC
b.	If $1 and $3 are not equal, PC + 4 If $1 and $3 are equal, PC + 4 + 4 $\times$ 12	PC to Add (PC + 4) to branch Mux, or PC to Add (PC + 4) to Add (adds offset) to branch Mux. After the branch Mux, we go through jump Mux and into the PC

4.11.4

	WrReg Mux	ALU Mux	Mem/ALU Mux	Branch Mux	Jump Mux
a.	3	16	0	PC + 4	PC + 4
b.	3 or 0 (RegDst is X)	− 3	X	PC + 4	PC + 4

4.11.5

	ALU	Add (PC + 4)	Add (Branch)
a.	2 and 16	PC and 4	PC + 4 and 16 × 4
b.	− 16 and − 3	PC and 4	PC + 4 and 12 × 4

4.11.6

	Read Register 1	Read Register 2	Write Register	Write Data	RegWrite
a.	2	3	3	0	1
b.	1	3	X (3 or 0)	X	0

Solution 4.12

4.12.1

	Pipelined	Single-cycle
a.	500ps	1650ps
b.	200ps	800ps

4.12.2

	Pipelined	Single-cycle
a.	2500ps	1650ps
b.	1000ps	800ps

4.12.3

	Stage to split	New clock cycle time
a.	MEM	400ps
b.	IF	190ps

4.12.4

a.	25%
b.	45%

4.12.5

a.	65%
b.	60%

4.12.6 We already computed clock cycle times for pipelined and single cycle organizations in 4.12.1, and the multi-cycle organization has the same clock cycle time as the pipelined organization. We will compute execution times relative to the pipelined organization. In single-cycle, every instruction takes one (long) clock cycle. In pipelined, a long-running program with no pipeline stalls completes one instruction in every cycle. Finally, a multi-cycle organization completes a   lw in 5 cycles, a sw in 4 cycles (no WB), an ALU instruction in 4 cycles (no MEM), and a beq in 4 cycles (no WB). So we have the speed-up of pipeline

	Multi-cycle execution time is X times pipelined execution time, where X is	Single-cycle execution time is X times pipelined execution time, where X is
a.	$0.15 \times 5 + 0.85 \times 4 = 4.15$	1650ps/500ps = 3.30
b.	$0.30 \times 5 + 0.70 \times 4 = 4.30$	800ps/200ps = 4.00

Solution 4.13

4.13.1

	Instruction sequence	Dependences
a.	I1: lw $1,40($6) I2: add $6,$2,$2 I3: sw $6,50($1)	RAW on $1 from I1 to I3 RAW on $6 from I2 to I3 WAR on $6 from I1 to I2 and I3
b.	I1: lw $5,-16($5) I2: sw $5,-16($5) I3: add $5,$5,$5	RAW on $5 from I1 to I2 and I3 WAR on $5 from I1 and I2 to I3 WAW on $5 from I1 to I3

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

	Instruction sequence	

a.	lw $1,40($6)
	add $6,$2,$2
	nop
	sw $6,50($1)
		Delay I3 to avoid RAW hazard on $1 from I1

b.	lw $5,-16($5)
	nop
	nop
	sw $5,-16($5)
	add $5,$5,$5
		Delay I2 to avoid RAW hazard on $5 from I1
		
		Note: no RAW hazard from on $5 from I1 now

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

	Instruction sequence	

a.	lw $1,40($6)
	add $6,$2,$2
	sw $6,50($1)
		No RAW hazard on $1 from I1 (forwarded)

b.	lw $5,-16($5)
	nop
	sw $5,-16($5)
	add $5,$5,$5
		Delay I2 to avoid RAW hazard on $5 from I1
		Value for $5 is forwarded from I2 now
		Note: no RAW hazard from on $5 from I1 now

4.13.4 The total execution time is the clock cycle time times the number of cycles. Without any stalls, a three-instruction sequence executes in 7 cycles (5 to complete the ?rst instruction, then one per instruction). The execution without forwarding must add a stall for every nop we had in 4.13.2, and execution forwarding must add a stall cycle for every nop we had in 4.13.3. Overall, we get:

	No forwarding	With forwarding	Speed-up due to forwarding
a.	$(7 + 1) \times 300ps = 2400ps$	$7 \times 400ps = 2800ps$	0.86 (This is really a slowdown)
b.	$(7 + 2) \times 200ps = 1800ps$	$(7 + 1) \times 250ps = 2000ps$	0.90 (This is really a slowdown)

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

	Instruction sequence	

a.	lw $1,40($6)
	add $6,$2,$2
	nop
	sw $6,50($1)
		Can't use ALU-ALU forwarding, ($1 loaded in MEM)

b.	lw $5,-16($5)
	nop
	nop
	sw $5,-16($5)
	add $5,$5,$5
		Can't use ALU-ALU forwarding ($5 loaded in MEM)

4.13.6

	No forwarding	With ALU-ALU forwarding only	Speed-up with ALU-ALU forwarding
a.	(7 + 1) × 300ps = 2400ps	(7 + 1) × 360ps = 2880ps	0.83 (This is really a slowdown)
b.	(7 + 2) × 200ps = 1800ps	(7 + 2) × 220ps = 1980ps	0.91 (This is really a slowdown)

Solution 4.14

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

	Instruction	Pipeline stage	Cycles

a.	lw $1,40($6)
	beq $2,$0,Lbl
	add $2,$3,$4
	sw $3,50($4)
		IF ID EX MEM WB
		    IF ED EX MEM WB
		        IF ID EX MEM WB
		            *** IF ID EX MEM WB
			9

b.	lw $5,-16($5)
	sw $4,-16($4)
	lw $3,-20($4)
	beq $2,$0,Lbl
	add $5,$1,$4
		IF ID EX MEM WB
		    IF ED EX MEM WB
		        IF ID EX MEM WB
		            *** *** *** IF ID EX MEM WB
		                        IF ID EX MEM WB
			12

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

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

	Instructions Executed	Cycles with 5 stages	Cycles with 4 stages	Speed-up
a.	4	4 + 4 = 8	3 + 4 = 7	8/7 = 1.14
b.	5	4 + 5 = 9	3 + 5 = 8	9/8 = 1.13

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

	Instructions Executed	Branches Executed	Cycles with branch in EXE	Cycles with branch in ID	Speed-up
a.	4	1	4 + 4 + 1 × 2 = 10	4 + 4 + 1 × 1 = 9	10/9 = 1.11
b.	5	1	4 + 5 + 1 × 2 = 11	4 + 5 + 1 × 1 = 10	11/10 = 1.10

4.14.4   The number of cycles for the (normal) 5-stage and the (combined EX/ MEM) 4-stage pipeline is already computed in 4.14.2. The clock cycle time is equal to the latency of the longest-latency stage. Combining EX and MEM stages affects clock time only if the combined EX/MEM stage becomes the longest-latency stage:

	Cycle time with 5 stages	Cycle time with 4 stages	Speed-up
a.	130ps (MEM)	150ps (MEM + 20ps)	(8 × 130)/(7 × 150) = 0.99
b.	220ps (MEM)	240ps (MEM + 20ps)	(9 × 220)/(8 × 240) = 1.03

4.14.5

	New ID latency	New EX latency	New cycle time	Old cycle time	Speed-up
a.	180ps	80ps	180ps (ID)	130ps (MEM)	(10 × 130)/(9 × 180) = 0.80
b.	150ps	160ps	220ps (MEM)	220ps (MEM)	(11 × 220)/(10 × 220) = 1.10

4.14.6   The cycle time remains unchanged: a 20ps reduction in EX latency has no effect on clock cycle time because EX is not the longest-latency stage. The change

does affect execution time because it adds one additional stall cycle to each branch. Because the clock cycle time does not improve but the number of cycles increases, the speed-up from this change will be below 1 (a slowdown). In 4.14.3 we already computed the number of cycles when branch is in EX stage. We have:

	Cycles with branch in EX	Execution time (branch in EX)	Cycles with branch in MEM	Execution time (branch in MEM)	Speed-up
a.	4 + 4 + 1 × 2 = 10	10 × 130ps = 1300ps	4 + 4 + 1 × 3 = 11	11 × 130ps = 1430ps	0.91
b.	4 + 5 + 1 × 2 = 11	11 × 220ps = 2420ps	4 + 5 + 1 × 3 = 12	12 × 220ps = 2640ps	0.92

Solution 4.15

4.15.1

a.	This instruction behaves like a load with a zero offset until it fetches the value from memory. The pre-ALU Mux must have another input now (zero) to allow this. After the value is read from memory in the MEM stage, it must be compared against zero. This must either be done quickly in the WB stage, or we must add another stage between MEM and WB. The result of this zero-comparison must then be used to control the branch Mux, delaying the selection signal for the branch Mux until the WB stage.
b.	We need to compute the memory address using two register values, so the address computation for SWI is the same as the value computation for the ADD instruction. However, now we need to read a third register value, so Registers must be extended to support a another read register input and another read data output and a Mux must be added in EX to select the Data Memory's write data input between this value and the value for the normal SW instruction.

4.15.2

a.	We need to add one more bit to the control signal for the pre-ALU Mux. We also need a control signal similar to the existing " Branch" signal to control whether or not the new zero-compare result is allowed to change the PC.
b.	We need a control signal to control the new Mux in the EX stage.

4.15.3

a.	This instruction introduces a new control hazard. The new PC for this branch is computed only after the Mem stage. If a new stage is added after MEM, this either adds new forwarding paths (from the new stage to EX) or (if there is no forwarding) makes a stall due to a data hazard one cycle longer.
b.	This instruction does not affect hazards. It modi? es no registers, so it causes no data hazards. It is not a branch instruction, so it produces no control hazards. With the added third register read port, it creates no new resource hazards, either.

4.15.4

a.	lw Rtmp,0(Rs) beq Rt,$0,Label	E.g., BEZI can be used when trying to ? nd the length of a zero-terminated array.
b.	add Rtmp,Rs,Rt sw Rd,0(Rtmp)	E.g., SWI can be used to store to an array element, where the array begins at address Rt and Rs is used as an index into the array.

4.15.5   The instruction can be translated into simple MIPS-like micro-operations (see 4.15.4 for a possible translation). These micro-operations can then be executed by the processor with a   " normal "   pipeline.

4.15.6   We will compute the execution time for every replacement interval. The old execution time is simply the number of instruction in the replacement interval (CPI of 1). The new execution time is the number of instructions after we made the replacement, plus the number of added stall cycles. The new number of instructions is the number of instructions in the original replacement interval, plus the new instruction, minus the number of instructions it replaces:

	New execution time	Old execution time	Speed-up
a.	20 - (2 - 1) + 1 = 20	20	1.00
b.	60 - (3 - 1) + 0 = 58	60	1.03

Solution 4.16

4.16.1   For every instruction, the IF/ID register keeps the PC + 4 and the instruction word itself. The ID/EX register keeps all control signals for the EX, MEM, and WB stages, PC+ 4, the two values read from Registers, the sign-extended lowermost 16 bits of the instruction word, and Rd and Rt ?   elds of the instruction word (even for instructions whose format does not use these ? elds). The EX/MEM register keeps control signals for MEM and WB stages, the PC + 4 + Offset (where Offset is the sign-extended lowermost 16 bits of the instructions, even for instructions that have no offset ? eld), the ALU result and the value of its Zero output, the value that was read from the second register in the ID stage (even for instructions that never need this value), and the number of the destination register (even for instructions that need no register writes; for these instructions the number of the destination register is simply a   " random "   choice between Rd or Rt). The MEM/WB register keeps the WB control signals, the value read from memory (or a   " random " value if there was no memory read), the ALU result, and the number of the destination register.

4.16.2

	Need to be read	Actually read
a.	$6	$6, $1
b.	$5	$5 (twice)

4.16.3

	EX	MEM
a.	40 + $6	Load value from memory
b.	$5 + $5	Nothing

4.16.4

	Loop	
a.	2:add $5,$5,$8 2:add $6,$6,$8 2:sw $1,20($5) 2:beq $1,$0,Loop 3:lw $1,40($6) 3:add $5,$5,$8 3:add $6,$6,$8 3:sw $1,20($5) 3:beq $1,$0,Loop	WB MEMWB EX MEM WB ID EX MEM WB IF ID EX MEM WB IF ID EX MEM IF ID EX IF ID IF

b.	sw $0,0($1)
	sw $0,4($1)
	add $2,$2,$4
	beq $2,$0,Loop
	add $1,$2,$3
	sw $0,0($1)
	sw $0,4($1)
	add $2,$2,$4
	beq $2,$0,Loop
		WB
		MEM WB
		EX MEMWB
		ID EX MEM WB
		IF ID EX MEMWB
		  IF ID EX MEM
		      IF ID EX
		          IF ID
		                IF

4.16.5   In a particular clock cycle, a pipeline stage is not doing useful work if it is stalled or if the instruction going through that stage is not doing any useful work there. In the pipeline execution diagram from 4.16.4, a stage is stalled if its name is not shown for a particular cycles, and stages in which the particular instruction is not doing useful work are marked in red. Note that a BEQ instruction is doing useful work in the MEM stage, because it is determining the correct value of the next instruction   ' s PC in that stage. We have:

	Cycles per loop iteration	Cycles in which all stages do useful work	% of cycles in which all stages do useful work
a.	5	1	20%
b.	5	2	40%

4.16.6   The address of that ?rst instruction of the third iteration (PC   + 4 for the beq from the previous iteration) and the instruction word of the   beq from the previous iteration.

Solution 4.17

4.17.1   Of all these instructions, the value produced by this adder is actually used only by a beq instruction when the branch is taken. We have:

a.	15% (60% of 25%)
b.	9% (60% of 15%)

4.17.2   Of these instructions, only   add needs all three register ports (reads two registers and write one). beq and sw does not write any register, and lw only uses one register value. We have:

a.	50%
b.	30%

4.17.3   Of these instructions, only   lw   and sw use the data memory. We have:

a.	25% (15% + 10%)
b.	55% (35% + 20%)

4.17.4   The clock cycle time of a single-cycle is the sum of all latencies for the logic of all ?ve stages. The clock cycle time of a pipelined datapath is the maximum latency of the ?ve stage logic latencies, plus the latency of a pipeline register that keeps the results of each stage for the next stage. We have:

	Single-cycle	Pipelined	Speed-up
a.	500ps	140ps	3.57
b.	730ps	230ps	3.17

4.17.5   The latency of the pipelined datapath is unchanged (the maximum stage latency does not change). The clock cycle time of the single-cycle datapath is the

sum of logic latencies for the four stages (IF, ID, WB, and the combined EX + MEM stage). We have:

	Single-cycle	Pipelined
a.	410ps	140ps
b.	560ps	230ps

4.17.6　The clock cycle time of the two pipelines (5-stage and 4-stage) as explained for 4.17.5. The number of instructions increases for the 4-stage pipeline, so the speed-up is below 1 (there is a slowdown):

	Instructions with 5-stage	Instructions with 4-stage	Speed-up
a.	$1.00 \times I$	$1.00 \times I + 0.5 \times (0.15 + 0.10) \times I = 1.125 \times I$	$0.89 \times$
b.	$1.00 \times I$	$1.00 \times I + 0.5 \times (0.35 + 0.20) \times I = 1.275 \times I$	$0.78 \times$

Solution 4.18

4.18.1　No signals are asserted in IF and ID stages. For the remaining three stages we have:

	EX	MEM	WB
a.	ALUSrc = 0, ALUOp = 10, RegDst = 1	Branch = 0, MemWrite = 0, MemRead = 0	MemtoReg = 1, RegWrite = 1
b.	ALUSrc = 0, ALUOp = 10, RegDst = 1	Branch = 0, MemWrite = 0, MemRead = 0	MemtoReg = 1, RegWrite = 1

4.18.2　One clock cycle.

4.18.3　The PCSrc signal is 0 for this instruction. The reason against generating the PCSrc signal in the EX stage is that the and must be done after the ALU computes its Zero output. If the EX stage is the longest-latency stage and the ALU output is on its critical path, the additional latency of an AND gate would increase the clock cycle time of the processor. The reason in favor of generating this signal in the EX stage is that the correct next-PC for a conditional branch can be computed one cycle earlier, so we can avoid one stall cycle when we have a control hazard.

4.18.4

	Control signal 1	Control signal 2
a.	Generated in ID, used in EX	Generated in ID, used in WB
b.	Generated in ID, used in MEM	Generated in ID, used in WB

4.18.5

a.	R-type instructions
b.	Loads.

4.18.6   Signal 2 goes back though the pipeline. It affects execution of instructions that execute after the one for which the signal is generated, so it is not a time-travel paradox.

Solution 4.19

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

	CPI	Stall Cycles
a.	$1 + 0.45 \times 2 + 0.05 \times 1 = 1.95$	49% (0.95/1.95)
b.	$1 + 0.40 \times 2 + 0.10 \times 1 = 1.9$	47% (0.9/1.9)

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

	CPI	Stall Cycles
a.	$1 + 0.25 = 1.25$	20% (0.25/1.25)
b.	$1 + 0.20 = 1.20$	17% (0.20/1.20)

4.19.3   With forwarding only from the EX/MEM register, EX to 1st dependences can be satis?ed without stalls but EX to 2nd and MEM to 1st dependences incur a one-cycle stall. With forwarding only from the MEM/WB register, EX to 2nd dependences incur no stalls. MEM to 1st dependences still incur a one-cycle stall (no time travel), and EX to 1st dependences now incur one stall cycle because we must wait for the instruction to complete the MEM stage to be able to forward to the next instruction. We compute stall cycles per instructions for each case as follows:

	EX/MEM	MEM/WB	Fewer stall cycles with
a.	$0.10 + 0.05 + 0.25 = 0.40$	$0.10 + 0.10 + 0.25 = 0.45$	EX/MEM
b.	$0.05 + 0.10 + 0.20 = 0.35$	$0.15 + 0.05 + 0.20 = 0.40$	EX/MEM

4.19.4 In 4.19.1 and 4.19.2 we have already computed the CPI without forwarding and with full forwarding. Now we compute time per instruction by taking into account the clock cycle time:

	Without forwarding	With forwarding	Speed-up
a.	1.95 × 100ps = 195ps	1.25 × 110ps = 137.5ps	1.42
b.	1.90 × 300ps = 570ps	1.20 × 350ps = 420ps	1.36

4.19.5 We already computed the time per instruction for full forwarding in 4.19.4. Now we compute time-per instruction with time-travel forwarding and the speed-up over full forwarding:

	With full forwarding	Time-travel forwarding	Speed-up
a.	1.25 × 110ps = 137.5ps	1 × 210ps = 210ps	0.65
b.	1.20 × 350ps = 420ps	1 × 450ps = 450ps	0.93

4.19.6

	EX/MEM	MEM/WB	Shorter time per instruction with
a.	1.40 × 100ps = 140ps	1.45 × 100ps = 145ps	EX/MEM
b.	1.35 × 320ps = 432ps	1.40 × 310ps = 434ps	EX/MEM

Solution 4.20

4.20.1

	Instruction sequence	RAW	WAR	WAW
a.	I1: lw $1,40($2) I2: add $2,$3,$3 I3: add $1,$1,$2 I4: sw $1,20($2)	($1) I1 to I3 ($2) I2 to I3, I4 ($1) I3 to I4	($2) I1 to I2	($1) I1 to I3
b.	I1: add $1,$2,$3 I2: sw $2,0($1) I3: lw $1,4($2) I4: add $2,$2,$1	($1) I1 to I2 ($1) I3 to I4	($2) I1, I2, I3 to I4 ($1) I1, I2 to I3	($1) I1 to I3

4.20.2 Only RAW dependences can become data hazards. With forwarding, only RAW dependences from a load to the very next instruction become hazards.

Without forwarding, any RAW dependence from an instruction to one of the following three instructions becomes a hazard:

	Instruction sequence	With forwarding	Without forwarding
a.	I1: lw $1,40($2) I2: add $2,$3,$3 I3: add $1,$1,$2 I4: sw $1,20($2)		($1) I1 to I3 ($2) I2 to I3, I4 ($1) I3 to I4
b.	I1: add $1,$2,$3 I2: sw $2,0($1) I3: lw $1,4($2) I4: add $2,$2,$1	($1) I3 to I4	($1) I1 to I2 ($1) I3 to I4

4.20.3   With forwarding, only RAW dependences from a load to the next two instructions become hazards because the load produces its data at the end of the second MEM stage. Without forwarding, any RAW dependence from an instruction to one of the following 4 instructions becomes a hazard:

	Instruction sequence	With forwarding	RAW
a.	I1: lw $1,40($2) I2: add $2,$3,$3 I3: add $1,$1,$2 I4: sw $1,20($2)	($1) I1 to I3	($1) I1 to I3 ($2) I2 to I3, I4 ($1) I3 to I4
b.	I1: add $1,$2,$3 I2: sw $2,0($1) I3: lw $1,4($2) I4: add $2,$2,$1	($1) I3 to I4	($1) I1 to I2 ($1) I3 to I4

4.20.4

	Instruction sequence	RAW
a.	I1: lw $1,40($2) I2: add $2,$3,$3 I3: add $1,$1,$2 I4: sw $1,20($2)	($1) I1 to I3 (0 overrides 1) ($2) I2 to I3 (2000 overrides 31)
b.	I1: add $1,$2,$3 I2: sw $2,0($1) I3: lw $1,4($2) I4: add $2,$2,$1	($1) I1 to I2 (2563 overrides 63)

4.20.5   A register modi? cation becomes " visible " to the EX stage of the following instructions only two cycles after the instruction that produces the register value leaves the EX stage. Our forwarding-assuming hazard detection unit only adds a

one-cycle stall if the instruction that immediately follows a load is dependent on the load. We have:

	Instruction sequence with forwarding stalls	Execution without forwarding	Values after execution
a.	I1: lw $1,40($2) I2: add $2,$3,$3 I3: add $1,$1,$2 I4: sw $1,20($2)	$1 = 0 (I4 and after) $2 = 2000 (after I4) $1 = 32 (after I4)	$0 = 0 $1 = 32 $2 = 2000 $3 = 1000
b.	I1: add $1,$2,$3 I2: sw $2,0($1) I3: lw $1,4($2) Stall I4: add $2,$2,$1	$1 = 2563 (Stall and after) $1 = 0 (after I4) $2 = 2626 (after I4)	$0 = 0 $1 = 0 $2 = 2626 $3 = 2500

4.20.6

	Instruction sequence with forwarding stalls	Correct execution	Sequence with NOPs

a.	I1: lw $1,40($2)
	I2: add $2,$3,$3
	I3: add $1,$1,$2
	I4: sw $1,20($2)
		I1: lw $1,40($2)
		I2: add $2,$3,$3
		    Stall
		    Stall
		I3: add $1,$1,$2
		    Stall
		    Stall
		I4: sw $1,20($2)
			lw $1,40($2)
			add $2,$3,$3
			nop
			nop
			add $1,$1,$2
			nop
			nop
			sw $1,20($2)

b.	I1: add $1,$2,$3
	I2: sw $2,0($1)
	I3: lw $1,4($2)
	    Stall
	I4: add $2,$2,$1
		I1: add $1,$2,$3
		    Stall
		    Stall
		I2: sw $2,0($1)
		I3: lw $1,4($2)
		    Stall
		    Stall
		I4: add $2,$2,$1
			add $1,$2,$3
			nop
			nop
			sw $2,0($1)
			lw $1,4($2)
			nop
			nop
			add $2,$2,$1

Solution 4.21

4.21.1

a.	lw $1,40($6)
	nop
	nop
	add $2,$3,$1
	add $1,$6,$4
	nop
	sw $2,20($4)
	and $1,$1,$4

b.	add $1,$5,$3
	nop
	nop
	sw $1,0($2)
	lw $1,4($2)
	nop
	nop
	add $5,$5,$1
	sw $1,0($2)

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

a.	I1: lw $7,40($6)
	I3: add $1,$6,$4
	    nop
	I2: add $2,$3,$7
	I5: and $1,$1,$4
	    nop
	I4: sw $2,20($4)
		Produce $7 instead of $1
		Moved up to ? ll NOP slot
		
		Use $7 instead of $1
		Moved up to ? ll NOP slot

b.	I1: add $7,$5,$3 I3: lw $1,4($2) nop I2: sw $7,0($2) I4: add $5,$5,$1 I5: sw $1,0($2)	Produce $7 instead of $1 Moved up to ? ll NOP slot Use $7 instead of $1

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

a.	I2 gets the value of $1 from before I1, not from I1 as it should.
b.	I4 gets the value of $1 from I1, not from I3 as it should.

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

	Instruction sequence	First ? ve cycles 1 2 3 4 5	Signals

a.	lw $1,40($6)
	add $2,$3,$1
	add $1,$6,$4
	sw $2,20($4)
	and $1,$1,$4
		IF ID EX MEM WB
		  IF ID *** EX
		    IF *** ID
		        IF
			1: PCWrite = 1, ALUin1 = X, ALUin2 = X
			2: PCWrite = 1, ALUin1 = X, ALUin2 = X
			3: PCWrite = 1, ALUin1 = 0, ALUin2 = 0
			4: PCWrite = 0, ALUin1 = X, ALUin2 = X
			5: PCWrite = 1, ALUin1 = 0, ALUin2 = 2

b.	add $1,$5,$3
	sw $1,0($2)
	lw $1,4($2)
	add $5,$5,$1
	sw $1,0($2)
		IF ID EX MEM WB
		  IF ID EX MEM
		    IF ID EX
		      IF ID
		        IF
			1: PCWrite = 1, ALUin1 = X, ALUin2 = X
			2: PCWrite = 1, ALUin1 = X, ALUin2 = X
			3: PCWrite = 1, ALUin1 = 0, ALUin2 = 0
			4: PCWrite = 1, ALUin1 = 0, ALUin2 = 1
			5: PCWrite = 1, ALUin1 = 0, ALUin2 = 0

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

pipeline register. The Rt ? eld from the ID/EX register is already an input of the hazard detection unit in Figure 4.60.

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

4.21.6   As explained for 4.21.5, we only need to specify the value of the PCWrite signal, because IF/IDWrite is equal to PCWrite and the ID/EXzero signal is its opposite. We have:

	Instruction sequence	First ? ve cycles					Signals
		1	2	3	4	5	

a.	lw $1,40($6)
	add $2,$3,$1
	add $1,$6,$4
	sw $2,20($4)
	and $1,$1,$4
		IF	ID
			IF
				EX
				ID
				IF
					MEM
					***
					***
						WB
						***
						***
						***
							1: PCWrite = 1
							2: PCWrite = 1
							3: PCWrite = 1
							4: PCWrite = 0
							5: PCWrite = 0

b.	add $1,$5,$3
	sw $1,0($2)
	lw $1,4($2)
	add $5,$5,$1
	sw $1,0($2)
		IF	ID
			IF
				EX
				ID
				IF
					MEM
					***
					***
						WB
						***
						***
						***
							1: PCWrite = 1
							2: PCWrite = 1
							3: PCWrite = 1
							4: PCWrite = 0
							5: PCWrite = 0

Solution 4.22

4.22.1

	Executed Instructions	Pipeline Cycles													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
a.	lw $1,40($6) beq $2,$3,Label2 (T) beq $1,$2,Label1 (NT) sw $2,20($4) and $1,$1,$4	IF	ID IF	EX ID IF	MEM EX ID	WB MEM EX	WB MEM IF	WB ID IF	EX ID	MEM EX	WB MEM	WB	12	13	14
b.	add $1,$5,$3 sw $1,0($2) add $2,$2,$3 beq $2,$4,Label1 (NT) add $5,$5,$1 sw $1,0($2)	IF	ID IF	EX ID IF	MEM EX ID IF	WB MEM EX ID	WB MEM EX IF	WB MEM ID IF	WB EX ID	MEM EX	WB MEM	WB		13	14

4.22.2

	Executed Instructions	Pipeline Cycles												
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
a.	lw $1,40($6)	IF	ID	EX	MEM	WB								13	14
	beq $2,$3,Label2 (T)		IF	ID	EX	MEM	WB								
	add $1,$6,$4			IF	ID	EX	MEM	WB							
	beq $1,$2,Label1 (NT)				IF	ID	***	EX	MEM	WB					
	sw $2,20($4)					IF	***	ID	EX	MEM	WB				
	and $1,$1,$4								IF	ID	EX	MEM	WB		
b.	add $1,$5,$3	IF	ID	EX	MEM	WB								13	14
	sw $1,0($2)		IF	ID	EX	MEM	WB								
	add $2,$2,$3			IF	ID	EX	MEM	WB							
	beq $2,$4,Label1 (NT)				IF	ID	EX	MEM	WB						
	add $5,$5,$1					IF	ID	EX	MEM	WB					
	sw $1,0($2)							IF	ID	EX	MEM	WB			

4.22.3

a.	Label1: lw $1,40($6)
	seq $8,$2,$3
	bnez $8,Label2 ; Taken
	add $1,$6,$4
	Label2: seq $8,$1,$2
	bnez $8,Label1 ; Not taken
	sw $2,20($4)
	and $1,$1,$4

b.	add $1,$5,$3
	Label1: sw $1,0($2)
	add $2,$2,$3
	bez $8,$2,$4
	bnez $8,Label1 ; Not taken
	add $5,$5,$1
	sw $1,0($2)

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

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

4.22.5   For 4.22.1 we have already shows the pipeline execution diagram for the case when branches are executed in the EX stage. The following is the pipeline diagram when branches are executed in the ID stage, including new stalls due to data dependences described for 4.22.4:

| | Executed Instructions | Pipeline Cycles | | | | | | | | | | | | |
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
a.	lw $1,40($6)	IF	ID	EX	MEM	WB							12	13	14
	beq $2,$3,Label2 (T)		IF	ID	EX	MEM	WB								
	beq $1,$2,Label1 (NT)			IF	***	ID	EX	MEM	WB						
	sw $2,20($4)				IF	ID	EX	MEM	WB						
	and $1,$1,$4					IF	ID	EX	MEM	WB					
b.	add $1,$5,$3	IF	ID	EX	MEM	WB								13	14
	sw $1,0($2)		IF	ID	EX	MEM	WB								
	add $2,$2,$3			IF	ID	EX	MEM	WB							
	beq $2,$4,Label1 (NT)				IF	***	ID	EX	MEM	WB					
	add $5,$5,$1						IF	ID	EX	MEM	WB				
	sw $1,0($2)							IF	ID	EX	MEM	WB			

Now the speed-up can be computed as:

a.	11/10 = 1.1
b.	12/12 = 1

4.22.6   Branch instructions are now executed in the ID stage. If the branch instruction is using a register value produced by the immediately preceding instruction, as we described for 4.22.4 the branch must be stalled because the preceding instruction is in the EX stage when the branch is already using the stale register values in the ID stage. If the branch in the ID stage depends on an R-type instruction that is in the MEM stage, we need forwarding to ensure correct execution of the branch. Similarly, if the branch in the ID stage depends on an R-type of load instruction in the WB stage, we need forwarding to ensure correct execution of the branch. Overall, we need another forwarding unit that takes the same inputs as the one that forwards to the EX stage. The new forwarding unit should control two Muxes placed right before the branch comparator. Each Mux selects between the value read from Registers, the ALU output from the EX/MEM pipeline register, and the data value from the MEM/WB pipeline register. The complexity of the new forwarding unit is the same as the complexity of the existing one.

Solution 4.23

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

	Extra CPI
a.	3 × (1 − 0.40) × 0.15 = 0.27
b.	3 × (1 − 0.60) × 0.10 = 0.12

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

	Extra CPI
a.	3 × (1 − 0.60) × 0.15 = 0.18
b.	3 × (1 − 0.40) × 0.10 = 0.18

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

	Extra CPI
a.	3 × (1 − 0.80) × 0.15 = 0.090
b.	3 × (1 − 0.95) × 0.10 = 0.015

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

	CPI without conversion	CPI with conversion	Speed-up from conversion
a.	1 + 3 × (1 − 0.80) × 0.15 = 1.090	1 + 3 × (1 − 0.80) × 0.15 × 0.5 = 1.045	1.090/1.045 = 1.043
b.	1 + 3 × (1 − 0.95) × 0.10 = 1.015	1 + 3 × (1 − 0.95) × 0.10 × 0.5 = 1.008	1.015/1.008 = 1.007

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

	CPI without conversion	Cycles per original instruction with conversion	Speed-up from conversion
a.	1.090	1 + (1 + 3 × (1 − 0.80)) × 0.15 × 0.5 = 1.120	1.090/1.120 = 0.97
b.	1.015	1 + (1 + 3 × (1 − 0.95)) × 0.10 × 0.5 = 1.058	1.015/1.058 = 0.96

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

	Correctly predicted	Correctly predicted non-loop-back	Accuracy on non-loop-back branches
a.	B × 0.80	B × 0.00	(B × 0.00)/(B × 0.20) = 0.00 (00%)
b.	B × 0.95	B × 0.15	(B × 0.15)/(B × 0.20) = 0.75 (75%)

Solution 4.24

4.24.1

	Always-taken	Always not-taken
a.	3/4 = 75%	1/4 = 25%
b.	3/5 = 60%	2/5 = 40%

4.24.2

	Outcomes	Predictor value at time of prediction	Correct or Incorrect	Accuracy
a.	T, T, NT, T	0, 1, 2, 1	I, I, I, I	0%
b.	T, T, T, NT	0, 1, 2, 3	I, I, C, I	25%

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

	Outcomes	Predictor value at time of prediction	Correct or Incorrect (in steady state)	Accuracy in steady state
a.	T, T, NT, T	1st occurrence: 0, 1, 2, 1 2nd occurrence: 2, 3, 2, 3 3rd occurrence: 3, 3, 3, 2 4th occurrence: 3, 3, 3, 2	C, C, I, C	75%
b.	T, T, T, NT, NT	1st occurrence: 0, 1, 2, 3, 2 2nd occurrence: 1, 2, 3, 3, 2 3rd occurrence: 1, 2, 3, 3, 2	C, C, C, I, I	60%

4.24.4   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.

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

4.24.6   The predictor is the same as in 4.24.4, 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 ? rst 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.

Solution 4.25

4.25.1

	Instruction 1	Instruction 2
a.	Over?ow (EX)	Invalid target address (EX)
b.	Invalid data address (MEM)	No exceptions

4.25.2   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.

4.25.3   Instructions are fetched normally until the exception is detected. When the exception is detected, all instructions that are in the pipeline after the ? rst 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 ?st instruction of the exception handler.

4.25.4

	Handler address
a.	0xFFFFF000
b.	0x00000010

The ?rst instruction word from the handler address is fetched in the cycle after the one in which the original exception is detected. When this instruction is decoded in the next cycle, the processor detects that the instruction is invalid. This exception is treated just like a normal exception — it converts the instruction being fetched in that cycle into a nop and puts the address of the Invalid Instruction handler into the PC at the end of the cycle in which the Invalid Instruction exception is detected.

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

4.25.6   We need a special instruction that allows us to move a value from the (exception) Cause register to a general-purpose register. We must ?rst 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 ?nally jump to that handler.

Solution 4.26

4.26.1   All exception-related signals are 0 in all stages, except the one in which the exception is detected. For that stage, we show values of Flush signals for various stages, and also the value of the signal that controls the Mux that supplies the PC value.

	Stage	Signals
a.	EX	IF.Flush = ID.Flush = EX.Flush = 1, PCSel = Exc
b.	MEM	IF.Flush = ID.Flush = EX.Flush = MEM.Flush = 1, PCSel = Exc This exception is detected in MEM, so we added MEM.Flush

4.26.2   The signals stored in the ID/EX stage are needed to execute the instruction if there are no exceptions. Figure 4.66 does not show it, but exception conditions from various stages are also supplied as inputs to the Control unit. The signal that goes directly to EX is EX.Flush and it is based on these exception condition inputs, not on the opcode of the instruction that is in the ID stage. In particular, the EX.Flush signal becomes 1 when the instruction in the EX stage triggers an exception and must be prevented from completing.

4.26.3   The disadvantage is that the exception handler begins executing one cycle later. Also, an exception condition normally checked in MEM cannot be delayed into WB, because at that time the instruction is updating registers and cannot be prevented from doing so.

4.26.4   When over?ow is detected in EX, each exception results in a 3-cycle delay (IF, ID, and EX are cancelled). By moving over?ow into MEM, we add one more cycle to this delay. To compute the speed-up, we compute execution time per 100,000 instructions:

	Old clock cycle time	New clock cycle time	Old time per 100,000 instructions	New time per 100,000 instructions	Speed-up
a.	350ps	350ps	350ps × 100,003	350ps × 100,004	0.99999
b.	210ps	210ps	210ps × 100,003	210ps × 100,004	0.99999

4.26.5   Exception control (Flush) signals are not really generated in the EX stage. They are generated by the Control unit, which is drawn as part of the ID stage, but we could have a separate " Exception Control " unit to generate Flush signals and this unit is not really a part of any stage.

4.26.6   Flush signals must be generated one Mux time before the end of the cycle. However, their generation can only begin after exception conditions are generated. For example, arithmetic over?ow is only generated after the ALU operation in EX is complete, which is usually in the later part of the clock cycle. As a result, the Control unit actually has very little time to generate these signals, and they can easily be on the critical path that determines the clock cycle time.

Solution 4.27

4.27.1   When the invalid instruction (I3) is decoded, IF.Flush and ID.Flush signals are used to convert I3 and I4 into nops (marked with *). In the next cycle, in IF we fetch the ?rst instruction of the exception handler, in ID we have a nop (instead of I4, marked), in EX we have a nop (instead of I3), and I1 and I2 still continue through the pipeline normally:

	Branch and delay slot	Pipeline
a.	I1: beq $1,$0,Label I2: sw $6,50($1) I3: Invalid I4: Something I5: Handler	IF ID EX MEM WB 　 IF ID EX MEM 　　 IF ID *EX 　　　 IF *ID 　　　　 IF
b.	I1: beq $5,$0,Label I2: nor $5,$4,$3 I3: Invalid I4: Something I5: Handler	IF ID EX MEM WB 　 IF ID EX MEM 　　 IF ID *EX 　　　 IF *ID 　　　　 IF

4.27.2   When I2 is in the MEM stage, it triggers an exception condition that results in converting I2 and I5 into nops (I3 and I4 are already nops by then). In the next cycle, we fetch I6, which is the ?rst instruction of the exception handler for the exception triggered by I2.

	Branch and delay slot	Branch and delay slot

a.	I1: beq $1,$0,Label
	I2: sw $6,50($1)
	I3: Invalid
	I4: Something
	I5: Handler 1
	I6: Handler 2
		IF ID EX MEM WB
		  IF ID EX MEM *WB
		    IF ID *EX *ME
		      IF *ID *EX
		        IF *ID
		          IF

b.	I1: beq $5,$0,Label
	I2: nor $5,$4,$3
	I3: Invalid
	I4: Something
	I5: Handler 1
	I6: Handler 2
		IF ID EX MEM WB
		  IF ID EX MEM *WB
		    IF ID *EX *ME
		      IF *ID *EX
		        IF *ID
		          IF

4.27.3   The EPC is the PC + 4 of the delay slot instruction. As described in Section 4.9, the exception handler subtracts 4 from the EPC, so it gets the address of the instruction that generated the exception (I2, the delay slot instruction). If the exception handler decides to resume execution of the application, it will jump to the I2. Unfortunately, this causes the program to continue as if the branch was not taken, even if it was taken.

4.27.4   The processor cancels the store instruction and other instructions (from the "Invalid instruction" exception handler) fetched after it, and then begins fetching instructions from the invalid data address handler. A major problem here is that the new exception sets the EPC to the instruction address in the "Invalid instruction" handler, overwriting the EPC value that was already there (address for continuing the program). If the invalid data address handler repairs the problem and attempts to continue the program, the "Invalid instruction" handler will be executed. However, if it manages to repair the problem and wants to continue the program, the EPC it incorrect (it was overwritten before it could be saved). This is the reason why exception handlers must be written carefully to avoid triggering exceptions themselves, at least until they have safely saved the EPC.

4.27.5   Not for store instructions. If we check for the address over?ow in MEM, the store is already writing data to memory in that cycle and we can no longer "cancel" it. As a result, when the exception handler is called the memory is already changed by the store instruction, and the handler can not observe the state of the machine that existed before the store instruction.

4.27.6 We must add two comparators to the EX stage, one that compares the ALU result to WADDR, and another that compares the data value from Rt to WVAL. If one of these comparators detects equality and the instruction is a store, this triggers a " Watchpoint " exception. As discussed for 4.27.5, we cannot delay the comparisons until the MEM stage because at that time the store is already done and we need to stop the application at the point before the store happens.

Solution 4.28

4.28.1

a.	add $1,$0,$0
	Again: beq $1,$2,End
	    add $6,$3,$1
	    lw $7,0($6)
	    add $8,$4,$1
	    sw $7,0($8)
	    addi $1,$1,1
	    beq $0,$0,Again
	End:

b.	add $4,$0,$0
	Again: add $1,$4,$6
	    lw $2,0($1)
	    lw $3,1($1)
	    beq $2,$3,End
	    sw $0,0($1)
	    addi $4,$4,1
	    beq $0,$0,Again
	End:

4.28.2

	Instructions	Pipeline

a.	add $1,$0,$0
	beq $1,$2,End
	add $6,$3,$1
	lw $7,0($6)
	add $8,$4,$1
	sw $7,0($8)
	addi $1,$1,1
	beq $0,$0,Again
	beq $1,$2,End
	add $6,$3,$1
	lw $7,0($6)
	add $8,$4,$1
	sw $7,0($8)
	addi $1,$1,1
	beq $0,$0,Again
	beq $1,$2,End
		IF ID EX ME WB
		IF ID ** EX ME WB
		  IF ** ID EX ME WB
		  IF ** ID ** EX ME WB
		    IF ** ID EX ME WB
		    IF ** ID ** EX ME WB
		      IF ** ID EX ME WB
		      IF ** ID ** EX ME WB
		        IF ** ID EX ME WB
		        IF ** ID ** EX ME WB
		          IF ** ID EX ME WB
		          IF ** ID EX ME WB
		            IF ID EX ME WB
		            IF ID EX ME WB
		              IF ID EX ME WB
		              IF ID ** EX ME WB

b.	add $4,$0,$0
	add $1,$4,$6
	lw $2,0($1)
	lw $3,1($1)
	beq $2,$3,End
	sw $0,0($1)
	addi $4,$4,1
	bew $0,$0,Again
	add $1,$4,$6
	lw $2,0($1)
	lw $3,1($1)
	beq $2,$3,End
	sw $0,0($1)
	addi $4,$4,1
	bew $0,$0,Again
	add $1,$4,$6
	lw $2,0($1)
	lw $3,1($1)
	beq $2,$3,End
		IF ID EX ME WB
		IF ID ** EX ME WB
		  IF ** ID EX ME WB
		  IF ** ID ** EX ME WB
		    IF ** ID ** EX ME WB
		    IF ** ID ** EX ME WB
		      IF ** ID EX ME WB
		      IF ** ID ** EX ME WB
		        IF ** ID EX ME WB
		        IF ** ID ** EX ME WB
		          IF ** ID EX ME WB
		          IF ** ID ** ** EX ME WB
		            IF ** ** ID EX ME WB
		            IF ** ** ID EX ME WB
		              IF ID EX ME WB
		              IF ID ** EX ME WB
		                IF ** ID EX ME WB
		                IF ** ID ** EX ME WB
		                  IF ** ID ** EX ME WB

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

a.	add $1,$0,$0
	Again: beq $1,$2,End
	    add $6,$3,$1
	    add $8,$4,$1
	    lw $7,0($6)
	    addi $1,$1,1
	    sw $7,0($8)
	    beq $0,$0,Again
	End:	

b.	add $4,$0,$0
	Again: add $1,$4,$6
	    lw $2,0($1)
	    lw $3,1($1)
	    beq $2,$3,End
	    sw $0,0($1)
	    addi $4,$4,1
	    beq $0,$0,Again
	End:
		We have not changed anything. Note that the only instruction without dependences to or from the two loads is ADDI, and it cannot be moved above the branch (then the loop would exit with the wrong value for i).

4.28.4

	Instructions	Pipeline

a.	add $1,$0,$0
	beq $1,$2,End
	add $6,$3,$1
	add $8,$4,$1
	lw $7,0($6)
	addi $1,$1,1
	sw $7,0($8)
	beq $0,$0,Again
	beq $1,$2,End
	add $6,$3,$1
	add $8,$4,$1
	lw $7,0($6)
	addi $1,$1,1
	sw $7,0($8)
	beq $0,$0,Again
	beq $1,$2,End
		IF ID EX ME WB
		IF ID ** EX ME WB
		  IF ** ID EX ME WB
		  IF ** ID ** EX ME WB
		     IF ** ID EX ME WB
		     IF ** ID EX ME WB
		       IF ID EX ME WB
		       IF ID EX ME WB
		         IF ID EX ME WB
		         IF ID ** EX ME WB
		           IF ** ID EX ME WB
		           IF ** ID EX ME WB
		              IF ID EX ME WB
		              IF ID EX ME WB
		                IF ID EX ME WB
		                IF ID ** EX ME WB

b.	add $4,$0,$0
	add $1,$4,$6
	lw $2,0($1)
	lw $3,1($1)
	beq $2,$3,End
	sw $0,0($1)
	addi $4,$4,1
	bew $0,$0,Again
	add $1,$4,$6
	lw $2,0($1)
	lw $3,1($1)
	beq $2,$3,End
	sw $0,0($1)
	addi $4,$4,1
	bew $0,$0,Again
	add $1,$4,$6
	lw $2,0($1)
	lw $3,1($1)
	beq $2,$3,End
		IF ID EX ME WB
		IF ID ** EX ME WB
		  IF ** ID EX ME WB
		  IF ** ID ** EX ME WB
		     IF ** ID ** EX ME WB
		     IF ** ID ** EX ME WB
		       IF ** ID EX ME WB
		       IF ** ID ** EX ME WB
		         IF ** ID EX ME WB
		         IF ** ID ** EX ME WB
		           IF ** ID EX ME WB
		           IF ** ID ** ** EX ME WB
		              IF ** ** ID EX ME WB
		              IF ** ** ID EX ME WB
		                IF ID EX ME WB
		                IF ID ** EX ME WB
		                  IF ** ID EX ME WB
		                  IF ** ID ** EX ME WB
		                    IF ** ID ** EX ME WB

4.28.5

	CPI for 1-issue	CPI for 2-issue	Speed-up
a.	1 (no data hazards)	0.86 (12 cycles for 14 instructions). In even-numbered iterations the LW and the SW can execute in parallel with the next instruction.	1.16
b.	1.14 (8 cycles per 7 instructions). There is 1 stall cycle in each iteration due to a data hazard between LW and the next instruction (BEQ).	1 (14 cycles for 14 instruction). Neither LW instruction can execute in parallel with another instruction, and the BEQ after the second LW is stalled because it depends on the load. However, SW always executes in parallel with another instruction (alternating between BEQ and ADDI).	1.14

4.28.6

	CPI for 1-issue	CPI for 2-issue	Speed-up
a.	1	0.64 (9 cycles for 14 instructions). In odd-numbered iterations ADD and LW cannot execute in the same cycle because of a data dependence, and then ADD and SW have the same problem. The rest of the instructions can execute in pairs.	1.56
b.	1.14	0.86 (12 cycles for 14 instructions). In all iterations BEQ is stalled because it depends on the second LW. In odd-numbered BEQ and SW execute together, and so do ADDI and the last BEQ. In even-numbered iterations SW and ADDI execute together, and so do the last BEQ and the ? rst ADD of the next iteration.	1.33

Solution 4.29

4.29.1   Note that all register read ports are active in every cycle, so 4 register reads (2 instructions with 2 reads each) happen in every cycle. We determine the number of cycles it takes to execute an iteration of the loop and the number of useful reads, then divide the two. The number of useful register reads for an instruction is the number of source register parameters minus the number of registers that are for-warded from prior instructions. We assume that register writes happen in the ?   rst half of the cycle and the register reads happen in the second half.

	Loop	Pipeline stages	Useful reads	% Useful

a.	addi $5,$5,-4
	beq $5,$0,Loop
	lw $1,40($6)
	add $5,$5,$1
	sw $1,20($5)
	addi $6,$6,4
	addi $5,$5,-4
	beq $5,$0,Loop
		ID EX ME WB
		ID ** EX ME WB
		IF ** ID EX ME WB
		IF ** ID ** ** EX ME WB
		 IF ** ** ID EX ME WB
		 IF ** ** ID EX ME WB
		  IF ID EX ME WB
		  IF ID ** EX ME WB
			
			1
			0 ($1, $5 fw)
			1 ($5 fw)
			1
			0 ($5 fw)
			1 ($5 fw)
				17%
				(4/(6 × 4))

b.	addi $2,$2,4
	beq $2,$0,Loop
	add $1,$2,$3
	sw $0,0($1)
	addi $2,$2,4
	beq $2,$0,Loop
		ID EX ME WB
		ID ** EX ME WB
		IF ** ID EX ME WB
		IF ** ID ** EX ME WB
		 IF ** ID EX ME WB
		 IF ** ID ** EX ME WB
			
			1 ($2 fw)
			1 ($1 fw)
			1
			1 ($2 fw)
				25%
				(4/(4 × 4))

4.29.2   The utilization of read ports is lower with a wider-issue processor:

	Loop	Pipeline stages	Useful reads	% Useful

a.	addi $6,$6,4
	addi $5,$5,-4
	beq $5,$0,Loop
	lw $1,40($6)
	add $5,$5,$1
	sw $1,20($5)
	addi $6,$6,4
	addi $5,$5,-4
	beq $5,$0,Loop
		ID EX ME WB
		ID EX ME WB
		ID ** EX ME WB
		IF ** ID EX ME WB
		IF ** ID ** ** EX ME WB
		IF ** ID ** ** ** EX ME WB
		 IF ** ** ** ID EX ME WB
		 IF ** ** ** ID EX ME WB
		 IF ** ** ** ID ** EX ME WB
			
			0 ($6 fw)
			0 ($1, $5 fw)
			0 ($1, $5 fw)
			1
			0 ($5 fw)
			1 ($5 fw)
				5.6%
				(2/(6 × 6))

b.	sw $0,0($1)
	addi $2,$2,4
	beq $2,$0,Loop
	add $1,$2,$3
	sw $0,0($1)
	addi $2,$2,4
	beq $2,$0,Loop
	add $1,$2,$3
	sw $0,0($1)
	addi $2,$2,4
	beq $2,$0,Loop
	add $1,$2,$3
	sw $0,0($1)
	addi $2,$2,4
	beq $2,$0,Loop
		ID EX ME WB
		ID EX ME WB
		ID ** EX ME WB
		IF ** ID EX ME WB
		IF ** ID ** EX ME WB
		IF ** ID ** EX ME WB
		 IF ** ID EX ME WB
		 IF ** ID EX ME WB
		 IF ** ID ** EX ME WB
		  IF ** ID EX ME WB
		  IF ** ID ** EX ME WB
		  IF ** ID ** EX ME WB
		   IF ** ID EX ME WB
		   IF ** ID EX ME WB
		   IF ** ID ** EX ME WB
			
			1 ($2 fw)
			1 ($1 fw)
			0 ($2 fw)
			1 ($2 fw)
			1 ($2 fw)
			1 ($1 fw)
			1
			1 ($2 fw)
			1 ($2 fw)
			1 ($1 fw)
			0 ($2 fw)
			1 ($2 fw)
				21%
				(10/(8 × 6))

4.29.3

	2 ports used	3 ports used
a.	1 cycle out of 6 (16.7%)	Never (0%)
b.	4 cycles out of 8 (50%)	Never (0%)

4.29.4

	Unrolled and scheduled loop	Comment

a.	Loop: lw $10,40($6)
	        lw $1,44($6)
	        addi $5,$5,-8
	        addi $6,$6,8
	        add $11,$5,$10
	        add $5,$11,$1
	        sw $10,28($11)
	        sw $1,24($5)
	        beq $5,$0,Loop
		The only time this code is unable to execute two instructions in the same cycle is in even-numbered iterations of the unrolled loop when the two ADD instruction are fetched together but must execute in different cycles.

b.	Loop: add $1,$2,$3
	        addi $2,$2,8
	        sw $0,-8($1)
	        sw $0,-4($1)
	        beq $2,$0,Loop
		We are able to execute two instructions per cycle in every iteration of this loop, so we execute two iterations of the unrolled loop every 5 cycles.

4.29.5   We determine the number of cycles needed to execute two iterations of the original loop (one iteration of the unrolled loop). Note that we cannot use CPI in our speed-up computation because the two versions of the loop do not execute the same instructions.

	Original loop	Unrolled loop	Speed-up
a.	6 × 2 = 12	5	2.4
b.	4 × 2 = 8	2.5 (5/2)	3.2

4.29.6   On a pipelined processor the number of cycles per iteration is easily computed by adding together the number of instructions and the number of stalls. The only stalls occur when a lw instruction is followed immediately with a RAW-dependent instruction, so we have:

	Original loop	Unrolled loop	Speed-up
a.	(6 + 1) × 2 = 14	9	1.6
b.	4 × 2 = 8	5	1.6

Solution 4.30

4.30.1   Let p be the probability of having a mispredicted branch. Whenever we have an incorrectly predicted beq as the ?rst of the two instructions in a cycle (the probability of this event is p), we waste one issue slot (half a cycle) and another two entire cycles. If the ?rst instruction in a cycle is not a mispredicted beq but the

second one is (the probability of this is (1 - p)   p), we waste two cycles. Without these mispredictions, we would be able to execute 2 instructions per cycle. We have:

	CPI
a.	0.5 + 0.02 × 2.5 + 0.98 × 0.02 × 2 = 0.589
b.	0.5 + 0.05 × 2.5 + 0.95 × 0.05 × 2 = 0.720

4.30.2   Inability to predict a branch results in the same penalty as a mispredicted branch. We compute the CPI like in 4.30.1, but this time we also have a 2-cycle penalty if we have a correctly predicted branch in the ?   rst issue slot and another branch that would be correctly predicted in the second slot. We have:

	CPI with 2 predicted branches per cycle	CPI with 1 predicted branch per cycle	Speed-up
a.	0.589	0.5 + 0.02 × 2.5 + 0.98 × 0.02 × 2 + 0.18 × 0.18 × 2 = 0.654	1.11
b.	0.720	0.5 + 0.05 × 2.5 + 0.95 × 0.05 × 2 + 0.10 × 0.10 × 2 = 0.740	1.03

4.30.3   We have a one-cycle penalty whenever we have a cycle with two instructions that both need a register write. Such instructions are ALU and   lw instructions. Note that beq does not write registers, so stalls due to register writes and due to branch mispredictions are independent events. We have:

	CPI with 2 register writes per cycle	CPI with 1 register write per cycle	Speed-up
a.	0.589	0.5 + 0.02 × 2.5 + 0.98 × 0.02 × 2 + 0.70 × 0.70 × 1 = 1.079	1.83
b.	0.720	0.5 + 0.05 × 2.5 + 0.95 × 0.05 × 2 + 0.75 × 0.75 × 1 = 1.283	1.78

4.30.4   We have already computed the CPI with the given branch prediction accuracy, and we know that the CPI with ideal branch prediction is 0.5, so:

	CPI with given branch prediction	CPI with perfect branch prediction	Speed-up
a.	0.589	0.5	1.18
b.	0.720	0.5	1.44

4.30.5   The CPI with perfect branch prediction is now 0.25 (four instructions per cycle). A branch misprediction in the ?   rst issue slot of a cycle results in 2.75 penalty cycles (remaining issue slots in the same cycle plus 2 entire cycles), in the

second issue slot 2.5 penalty cycles, in the third slot 2.25 penalty cycles, and in the last (fourth) slot 2 penalty cycles. We have:

	CPI with given branch prediction	CPI with perfect branch prediction	Speed-up
a.	$0.25 + 0.02 \times 2.75 + 0.98 \times 0.02 \times 2.5 + 0.98^2 \times 0.02 \times 2.25 + 0.98^3 \times 0.02 \times 2 = 0.435$	0.25	1.74
b.	$0.25 + 0.05 \times 2.75 + 0.95 \times 0.05 \times 2.5 + 0.95^2 \times 0.05 \times 2.25 + 0.95^3 \times 0.05 \times 2 = 0.694$	0.25	2.77

The speed-up from improved branch prediction is much larger in a 4-issue processor than in a 2-issue processor. In general, processors that issue more instructions per cycle gain more from improved branch prediction because each branch misprediction costs them more instruction execution opportunities (e.g., 4 per cycle in 4-issue versus 2 per cycle in 2-issue).

4.30.6   With this pipeline, the penalty for a mispredicted branch is 20 cycles plus the fraction of a cycle due to discarding instructions that follow the branch in the same cycle. We have:

	CPI with given branch prediction	CPI with perfect branch prediction	Speed-up
a.	$0.25 + 0.02 \times 20.75 + 0.98 \times 0.02 \times 20.5 + 0.98^2 \times 0.02 \times 20.25 + 0.98^3 \times 0.02 \times 20 = 1.832$	0.25	7.33
b.	$0.25 + 0.05 \times 20.75 + 0.95 \times 0.05 \times 20.5 + 0.95^2 \times 0.05 \times 20.25 + 0.95^3 \times 0.05 \times 20 = 4.032$	0.25	16.13

We observe huge speed-ups when branch prediction is improved in a processor with a very deep pipeline. In general, processors with deeper pipelines benefit more from improved branch prediction because these processors cancel more instructions (e.g., 20 stages worth of instructions in a 50-stage pipeline versus 2 stages worth of instructions in a 5-stage pipeline) on each misprediction.

Solution 4.31

4.31.1   The number of cycles is equal to the number of instructions (one instruction is executed per cycle) plus one additional cycle for each data hazard which occurs when a lw instruction is immediately followed by a dependent instruction. We have:

	CPI
a.	$(8 + 1)/8 = 1.13$
b.	$(7 + 1)/7 = 1.14$

4.31.2   The number of cycles is equal to the number of instructions (one instruction is executed per cycle), plus the stall cycles due to data hazards. Data

hazards occur when the memory address used by the instruction depends on the result of a previous instruction (EXE to ARD, 2 stall cycles) or the instruction after that (1 stall cycle), or when an instruction writes a value to memory and one of the next two instructions reads a value from the same address (2 or 1 stall cycles). All other data dependences can use forwarding to avoid stalls. We have:

	Instructions	Stall Cycles	CPI
a.	I1: mov -4(esp), eax I2: add (edx), eax I3: mov eax, -4(esp) I4: add 1, ecx I5: add 4, edx I6: cmp esi, ecx I7: jl Label	No stalls.	7/7 = 1
b.	I1: add eax, (edx) I2: mov eax, edx I3: add 1, eax I4: jl Label	No stalls.	4/4 = 1

4.31.3   The number of instructions here is that from the x86 code, but the number of cycles per iteration is that from the MIPS code (we fetch x86 instructions, but after instructions are decoded we end up executing the MIPS version of the loop):

	CPI
a.	9/7 = 1.29
b.	8/4 = 2

4.31.4   Dynamic scheduling allows us to execute an independent " future " instruction when the one we should be executing stalls. We have:

	Instructions	Reordering	CPI
a.	I1: lw $2,-4($sp) I2: lw $3,0($4) I3: add $2,$2,$3 I4: sw $2,-4($sp) I5: addi $6,$6,1 I6: addi $4,$4,4 I7: slt $1,$6,$5 I8: bne $1,$0,Label	I3 stalls, but we do I5 instead.	1 (no stalls)
b.	I1: lw $2,0($4) I2: add $2,$2,$5 I3: sw $2,0($4) I4: add $4,$5,$0 I5: addi $5,$5,1 I6: slt $1,$5,$0 I7: bne $1,$0,Label	I2 stalls, and all subsequent instructions have dependences so this stall remains.	(7 + 1)/7 = 1.14

4.31.5 We use t0, t1, etc. as names for new registers in our renaming. We have:

	Instructions	Stalls	CPI
a.	I1: lw t1,-4($sp) I2: lw $3,0($4) I3: add $2,t1,$3 I4: sw $2,-4($sp) I5: addi $6,$6,1 I6: addi $4,$4,4 I7: slt $1,$6,$5 I8: bne $1,$0,Label	I3 would stall, but I5 is executed instead.	1 (no stalls)
b.	I1: lw t1,0($4) I2: add $2,t1,$5 I3: sw $2,0($4) I4: add $4,$5,$0 I5: addi $5,$5,1 I6: slt $1,$5,$0 I7: bne $1,$0,Label	I2 stalls, and all subsequent instructions have dependences so this stall remains. Note that I4 or I5 cannot be done instead of I2 because of WAR dependences that are not eliminated. Renaming $4 in I4 or $5 in I5 does not eliminate any WAR dependences. This is a problem when renaming is done on the code (e.g., by the compiler). If the processor was renaming registers at runtime each instance of I4 would get a new name for the $4 it produces and we would be able to "cover" the I2 stall.	(7 + 1)/7 = 1.14

4.31.6 Note that now every time we execute an instruction it can be renamed differently. We have:

	Instructions	Reordering	CPI
a.	I1: lw t1,-4($sp) I2: lw t2,0($4) I3: add t3,t1,t2 I4: sw t3,-4($sp) I5: addi t4,$6,1 I6: addi t5,$4,4 I7: slt t6,t4,$5 I8: bne t6,$0,Label In next iteration uses of $6 renamed to t4, $4 renamed to t5.	No stalls remain. I3 would stall stalls, but we can do I5 instead.	1 (no stalls)
b.	I1: lw t1,0($4) I2: add t2,t1,$5 I3: sw t2,0($4) I4: add t3,$5,$0 I5: addi t4,$5,1 I6: slt t5,t4,$0 I7: bne t5,$0,Label In next iteration uses of $4 renamed to t3, $5 renamed to t4.	No stalls remain. I2 would stall, but we can do I4 instead.	7/7 = 1

Solution 4.32

4.32.1   The expected number of mispredictions per instruction is the probability that a given instruction is a branch that is mispredicted. The number of instructions between mispredictions is one divided by the number of mispredictions per instruction. We get:

	Mispredictions per instruction	Instructions between mispredictions
a.	$0.2 \times (1 - 0.9)$	50
b.	$0.20 \times (1 - 0.995)$	1000

4.32.2   The number of in-progress instructions is equal to the pipeline depth times the issue width. The number of in-progress branches can then be easily computed because we know what percentage of all instructions are branches. We have:

	In-progress branches
a.	$12 \times 4 \times 0.20 = 9.6$
b.	$25 \times 4 \times 0.20 = 20$

4.32.3   We keep fetching from the wrong path until the branch outcome is known, fetching 4 instructions per cycle. If the branch outcome is known in stage N of the pipeline, all instructions are from the wrong path in N - 1 stages. In the Nth stage, all instructions after the branch are from the wrong path. Assuming that the branch is just as likely to be the 1st, 2nd, 3rd or 4th instruction fetched in its cycle, we have on average 1.5 instructions from the wrong path in the Nth stage (3 is branch is 1st, 2 is branch is 2nd, 1 is branch is 3rd, and 0 if branch is last). We have:

	Wrong-path instructions
a.	$(10 - 1) \times 4 \times 1.5 = 37.5$
b.	$(18 - 1) \times 4 \times 1.5 = 69.5$

4.32.4   We can compute the CPI for each processor, then compute the speed-up. To compute the CPI, we note that we have determined the number of useful instructions between branch mispredictions (for 4.32.1) and the number of mis-fetched instructions per branch misprediction (for 4.32.3), and we know how many instructions in total are fetched per cycle (4 or 8). From that we can determine the

number of cycles between branch mispredictions, and then the CPI (cycles per useful instruction).   We have:

	4-issue		8-issue			Speed-up
	Cycles	CPI	Mis-fetched	Cycles	CPI	
a.	(37.5 + 50)/4 = 21.9	21.9/50 = 0.438	(10 − 1) × 8 × 3.5 = 75.5	(75.5 + 50)/8 = 15.7	15.7/50 = 0.314	1.39
b.	(69.5 + 1000)/4 = 267.4	267.4/1000 = 0.267	(18 − 1) × 8 × 3.5 = 139.5	(139.5 + 1000)/8 = 142.4	142.4/1000 = 0.142	1.88

4.32.5   When branches are executed one cycle earlier, there is one less cycle needed to execute instructions between two branch mispredivctions. We have:

	" Normal" CPI	" Improved" CPI	Speed-up
a.	21.9/50 = 0.438	20.9/50 = 0.418	1.048
b.	267.4/1000 = 0.267	266.4/1000 = 0.266	1.004

4.32.6

	" Normal" CPI	" Improved" CPI	Speed-up
a.	15.7/50 = 0.314	14.7/50 = 0.294	1.068
b.	142.4/1000 = 0.142	141.4/1000 = 0.141	1.007

Speed-ups from this improvement are larger for the 8-issue processor than with the 4-issue processor. This is because the 8-issue processor needs fewer cycles to execute the same number of instructions, so the same 1-cycle improvement represents a large relative improvement (speed-up).

Solution 4.33

4.33.1   We need two register reads for each instruction issued per cycle:

	Read ports
a.	4 × 2 = 8
b.	2 × 2 = 4

4.33.2   We compute the time-per-instruction as CPI times the clock cycle time. For the 1-issue 5-stage processor we have a CPI of 1 and a clock cycle time of T. For an N-issue K-stage processor we have a CPI of 1/N and a clock cycle of T   5/K. Overall, we get a speed-up of:

	Speed-up
a.	10/5 × 4 = 8
b.	25/5 × 2 = 10

4.33.3 We are unable to bene?t from a wider issue width (CPI is 1), so we have:

	Speed-up
a.	10/5 = 2
b.	25/5 = 5

4.33.4 We ?rst compute the number of instructions executed between mispredicted branches. Then we compute the number of cycles needed to execute these instructions if there were no misprediction stalls, and the number of stall cycles due to a misprediction. Note that the number of cycles spent on a misprediction in is the number of entire cycles (one less than the stage in which branches are executed) and a fraction of the cycle in which the mispredicted branch instruction is. The fraction of a cycle is determined by averaging over all possibilities. In an N-issue processor, we can have the branch as the ?rst instruction of the cycle, in which case we waste (N - 1) Nths of a cycle, or the branch can be the second instruction in the cycle, in which case we waste (N - 2) Nths of a cycle,   … , or the branch can be the last instruction in the cycle, in which case none of that cycle is wasted. With all of this data we can compute what percentage of all cycles are misprediction stall cycles:

	Instructions between branch mispredictions	Cycles between branch mispredictions	Stall Cycles	% Stalls
a.	1/(0.30 × 0.05) = 66.7	66.7/4 = 16.7	6.4	6/(16.7 + 6.4) = 26%
b.	1/(0.15 × 0.03) = 222.2	222.2/2 = 111.1	7.3	7/(111.1 + 7.3) = 5.9%

4.33.5 We have already computed the number of stall cycles due to a branch misprediction, and we know how to compute the number of non-stall cycles between mispredictions (this is where the misprediction rate has an effect). We have:

	Stall cycles between mispredictions	Need # of instructions between mispredictions	Allowed branch misprediction rate
a.	6.4	6.4 × 4/0.10 = 255	1/(255 × 0.30) = 1.31%
b.	7.3	7.3 × 2/0.02 = 725	1/(725 × 0.15) = 0.92%

The needed accuracy is 100% minus the allowed misprediction rate.

4.33.6   This problem is very similar to We have already computed the number of stall cycles due to a branch misprediction, and we know how to compute the number of non-stall cycles between mispredictions (this is where the misprediction rate has an effect). We have:, except that we are aiming to have as many stall cycles as we have non-stall cycles. We get:

	Stall cycles between mispredictions	Need # of instructions between mispredictions	Allowed branch misprediction rate
a.	6.4	$6.4 \times 4 = 25.5$	$1/(25.5 \times 0.30) = 13.1\%$
b.	7.3	$7.3 \times 2 = 14.5$	$1/(14.5 \times 0.15) = 46.0\%$

The needed accuracy is 100% minus the allowed misprediction rate.

Solution 4.34

4.34.1   We need an IF pipeline stage to fetch the instruction. Since we will only execute one kind of instruction, we do not need to decode the instruction but we still need to read registers. As a result, we will need an ID pipeline stage although it would be misnamed. After that, we have an EXE stage, but this stage is simpler because we know exactly which operation should be executed so there is no need for an ALU that supports different operations. Also, we need no Mux to select which values to use in the operation because we know exactly which value it will be. We have:

a.	In the ID stage we read two registers and we do not need a sign-extend unit. In the EXE stage we need an Add unit whose inputs are the two register values read in the ID stage. After the EXE stage we have a WB stage which writes the result from the Add unit into Rd (again, no Mux). Note that there is no MEM stage, so this is a 4-stage pipeline. Also note that the PC is always incremented by 4, so we do not need the other Add and Mux units that compute the new PC for branches and jumps.
b.	We only read one register in the ID stage so there is no need for the second read port in the Registers unit. We do need a sign-extend unit for the Offs ? eld in the instruction word. In the EXE stage we need an Add unit whose inputs are the register value and the sign-extended offset from the ID stage. After the EXE stage we use the output of the Add unit as a memory address in the MEM stage, and then we have a WB stage which writes the value we read in the MEM stage into Rt (again, no Mux). Also note that the PC is always incremented by 4, so we do not need the other Add and Mux units that compute the new PC for branches and jumps.

4.34.2

a.	Assuming that the register write in WB happens in the ? rst half of the cycle and the register reads in ID happen in the second half, we only need to forward the Add result from the EX/WB pipeline register to the inputs of the Add unit in the EXE stage of the next instruction (if that next instruction depends on the previous one). No hazard detection unit is needed because forwarding eliminates all hazards.
b.	Assuming that the register write in WB happens in the ? rst half of the cycle and the register read in ID happens in the second half, we only need to forward the memory value from the MEM/WB pipeline register to the ? rst (register) input of the Add unit in the EXE stage of the next or second-next instruction (if one of those two instructions is dependent on the one that has just read the value). We also need a hazard detection unit that stalls any instruction whose Rs register ? eld is equal to the Rt ? eld of the previous instruction.

4.34.3   We need to add some decoding logic to our ID stage. The decoding logic must simply check whether the opcode and funct ?   led (if there is a funct ?   eld) match this instruction. If there is no match, we must put the address of the exception handler into the PC (this adds a Mux before the PC) and ?   ush (convert to nops) the unde? ned instruction (write zeros to the ID/EX pipeline register) and the following instruction which has already been fetched (write zeros to the IF/ID pipeline register).

4.34.4

a.	We need to add the logic that computes the branch address (sign-extend, shift-left – 2, Add, and Mux to select the PC). We also need to replace the Add unit in EXE with an ALU that supports either an ADD or a comparison. The ALUOp signal to select between these operations must be supplied by the Control unit.
b.	We need to add back the second register read port (AND reads two registers), add the Mux that selects the value supplied to the second ALU input (register for AND, Offs for LW), add an ALUOp signal to select between two ALU operations, and replace the Add unit in EXE with an ALU that supports either an Add or an And operation. Finally, we must add to the WB stage the Mux that select whether the value to write to the register is the value from the ALU of from memory, and the Mux in the EX stage that selects which register to write to (Rd for AND, Rt for LW).

4.34.5

a.	The same forwarding logic used for forwarding from one ADD to another can also be used to forward from ADD to BEQ. We still need no hazard detection for data hazards, but we must add detection of control hazards. Assuming there is no branch prediction, whenever a BEQ is taken we must ? ush (convert to NOPs) all instructions that were fetched after that branch.
b.	We need to add forwarding from the EX/MEM pipeline register to the ALU inputs in the EXE stage (so AND can forward to the next instruction), and we need to extend our forwarding from the MEM/WB pipeline register to the second input of the ALU unit (so LW can forward to an AND whose Rt (input) register is the same as the Rt (result) register of the LW instruction. We also need to extend the hazard detection unit to also stall any AND instruction whose Rs or Rt register ? eld is equal to the Rt ? eld of the previous LW instruction.

4.34.6 The decoding logic must now check if the instruction matches either of the two instructions. After that, the exception handling is the same as for 4.34.3.

Solution 4.35

4.35.1 The worst case for control hazards is if the mispredicted branch instruction is the last one in its cycle and we have been fetching the maximum number of instructions in each cycle. Then the control hazard affects the remaining instructions in the branch's own pipeline stage and all instructions in stages between fetch and branch execution stage. We have:

	Delay slots needed
a.	$7 \times 4 - 1 = 27$
b.	$17 \times 2 - 1 = 33$

4.35.2 If branches are executed in stage X, the number of stall cycles due to a misprediction is (N - 1). These cycles are reduced by ?lling them with delay slot instructions. We compute the number of execution (non-stall) cycles between mis-predictions, and the speed-up as follows:

	Non-stall cycles between mispredictions	Stall cycles without delay slots	Stall cycles with 4 delay slots	Speed-up due to delay slots
a.	$1/(020 \times (1 - 0.80) \times 4) = 6.25$	6	5	$(6.25 + 6)/(6.25 + 5) = 1.089$
b.	$1/(025 \times (1 - 0.92) \times 2) = 25$	16	14	$(25 + 16)/(25 + 14) = 1.051$

4.35.3 For 20% of branches, we add an extra instruction, for 30% of the branches we add two extra instructions, and for 40% of branches, we add three extra instructions. Overall, an average branch instruction is now accompanied by $0.20 + 0.30 \times 2 + 0.40 \times 3 = 2$ nop instructions. Note that these nops are added for every branch, not just mispredicted ones. These nop instructions add to the execution time of the program, so we have:

	Total cycles between mispredictions without delay slots	Stall cycles with 4 delay slots	Extra cycles spent on NOPs	Speed-up due to delay slots
a.	$6.25 + 6 = 12.25$	5	$0.5 \times 6.25 \times 0.20 = 0.625$	$12.5/(6.25 + 5 + 0.625) = 1.032$
b.	$25 + 16 = 41$	14	$1 \times 25 \times 0.25 = 6.25$	$41/(25 + 14 + 6.25) = 0.906$

4.35.4

| a. | ```
add        $2,$0,$0          ;   $1=0
Loop: beq    $2,$3,End
        lb  $10,1000($2)      ;   Delay slot
        sb  $10,2000($2)
  beq          $0,$0,Loop
        addi $2,$2,1          ;   Delay slot
Exit:
``` |
|---|---|
| b. | ```
add        $2,$0,$0          ;   $1=0
Loop: lb      $10,1000($2)
        lb  $11,1001($2)
  beq          $10,$11,End
        addi $1,$1,1          ;   Delay slot
  beq          $0,$0,Loop
        addi $2,$2,1          ;   Delay slot
Exit:   addi $1,$1,-1         ;   Undo c++ from delay slot
``` |

4.35.5

| a. | ```
add        $2,$0,$0          ;   $1=0
Loop: beq    $2,$3,End
        lb  $10,1000($2)      ;   Delay slot
  nop                         ;   2nd delay slot
  beq          $0,$0,Loop
        sb  $10,2000($2)      ;   Delay slot
  addi         $2,$2,1        ;   2nd delay slot
Exit:
``` |
|---|---|
| b. | ```
add        $2,$0,$0          ;   $1=0
        lb  $10,1000($2)      ;   Prepare for ?    rst iteration
        lb  $11,1001($2)      ;   Prepare for ?    rst iteration
Loop: beq    $10,$11,End
        addi $1,$1,1          ;   Delay slot
  addi         $2,$2,1        ;   2nd delay slot
  beq          $0,$),Loop
        lb  $10,1000($2)      ;   Delay slot, prepare for next iteration
        lb  $11,1001($2)      ;   2nd delay slot, prepare for next iteration
Exit:   addi $1,$1,-1         ;   Undo c++ from delay slot
        addi $2,$2,-1         ;   Undo i++ from 2        nd delay slot
``` |
$$\text{(note: superscript } nd \text{ on last line)}$$

4.35.6   The maximum number of in-?   ight instructions is equal to the pipeline depth times the issue width. We have:

	Instructions in ? ight	Instructions per iteration	Iterations in ? ight
a.	$10 \times 4 = 40$	5	$40/5 + 1 = 9$
b.	$25 \times 2 = 50$	6	$\text{roundUp}(50/6) + 1 = 10$

Note that an iteration is in-ﬂight when even one of its instructions is in-ﬂight. This is why we add one to the number we compute from the number of instructions in ﬂight (instead of having an iteration entirely in ﬂight, we can begin another one and still have the "trailing" one partially in ﬂight, and you round up.

Solution 4.36

4.36.1

	Instruction	Translation
a.	lwinc Rt,Offset(Rs)	lw Rt,Offset(Rs) addi Rs,Rs,4
b.	addr Rt,Offset(Rs)	lw tmp,Offset(Rs) add Rt,Rt,tmp

4.36.2 The ID stage of the pipeline would now have a lookup table and a micro-PC, where the opcode of the fetched instruction would be used to index into the lookup table. Micro-operations would then be placed into the ID/EX pipeline register, one per cycle, using the micro-PC to keep track of which micro-op is the next one to be output. In the cycle in which we are placing the last micro-op of an instruction into the ID/EX register, we can allow the IF/ID register to accept the next instruction. Note that this results in executing up to one micro-op per cycle, but we actually fetching instructions less often than that.

4.36.3

	Instruction
a.	We need to add an incrementer in the MEM stage. This incrementer would increment the value read from Rs while memory is being accessed. We also need to change the Registers unit to allow two writes to happen in the same cycle, so we can write the value from memory into Rt and the incremented value of Rs back into Rs.
b.	We need another EX stage after the MEM stage to perform the addition. The result can then be stored into Rt in the WB stage.

4.36.4 Not often enough to justify the changes we need to make to the pipeline. Note that these changes slow down all the other instructions, so we are speeding up a relatively small fraction of the execution while slowing down everything else.

4.36.5 Each original addm instruction now results in executing two more instructions, and also adds a stall cycle (the add depends on the lw). As a result,

each cycle in which we executed an addm instruction now adds three more cycles to the execution. We have:

	Speed-up from addm translation
a.	1/(1 + 0.05 × 3) = 0.87
b.	1/(1 + 0.10 × 3) = 0.77

4.36.6 Each translated addm adds the 3 stall cycles, but now half of the existing stalls are eliminated. We have:

	Speed-up from addm translation
a.	1/(1 + 0.05 × 3 − 0.05/2) = 0.89
b.	1/(1 + 0.10 × 3 − 0.10/2) = 0.8

Solution 4.37

4.37.1   All of the instructions use the instruction memory, the PC + 4 adder, the control unit (to decode the instruction), and the ALU. For the least utilized unit, we have:

a.	The result of the branch adder (add offset to PC + 4) is only used by the BEQ instruction, the data memory read port is only used by the LW instruction, and the write port is only used by the last SW instruction (the ? rst SW is not executed because the BEW is taken).
b.	The result of the branch adder (add offset to PC + 4) is never used.

Note that the branch adder performs its operation in every cycle, but its result is actually used only when a branch is taken.

4.37.2   The read port is only used by lw and the write port by sw instructions. We have:

	Data memory read	Data memory write
a.	25% (1 out of 4)	25% (1 out of 4)
b.	40% (2 out of 5)	20% (1 out of 5)

4.37.3   In the IF/ID pipeline register, we need 32 bits for the instruction word and 32 bits for PC + 4 for a total of 64 bits. In the ID/EX register, we need 32 bits for each of the two register values, the sign-extended offset/immediate value, and PC + 4 (for exception handling). We also need 5 bits for each of the three register ? elds from the instruction word (Rs, Rt, Rd), and 10 bits for all the control signals output by the Control unit. The total for the ID/EX register is 153 bits.

In the EX/MEM register, we need 32 bits each for the value of register Rt and for the ALU result. We also need 5 bits for the number of the destination register and 4 bits for control signals. The total for the EX/MEM register is 73 bits. Finally, for the MEM/WB register we need 32 bits each for the ALU result and value from memory, 5 bits for the number of the destination register, and 2 bits for control signals. The total for MEM/WB is 71 bits. The grand total for all pipeline registers is 361 bits.

4.37.4   In the IF stage, the critical path is the I-Mem latency. In the ID stage, the critical path is the latency to read Regs. In the EXE stage, we have a Mux and then ALU latency. In the MEM stage we have the D-Mem latency, and in the WB stage we have a Mux latency and setup time to write Regs (which we assume is zero). For a single-cycle design, the clock cycle time is the sum of these per-stage latencies (for a load instruction). For a pipelined design, the clock cycle time is the longest of the per-stage latencies. To compare these clock cycle times, we compute a speed-up based on clock cycle time alone (assuming the number of clock cycles is the same in single-cycle and pipelined designs, which is not true). We have:

	IF	ID	EX	MEM	WB	Single-cycle	Pipelined	" Speed-up"
a.	400ps	200ps	150ps	350ps	30ps	1130ps	400ps	2.83
b.	500ps	220ps	280ps	1000ps	100ps	2100ps	1000ps	2.10

Note that this speed-up is signi? cantly lower than 5, which is the " ideal " speed of 5-stage pipelining.

4.37.5   If we only support add instructions, we do not need the MUX in the WB stage, and we do not need the entire MEM stage. We still need Muxes before the ALU for forwarding. We have:

	IF	ID	EX	WB	Single-cycle	Pipelined	" Speed-up"
a.	400ps	200ps	150ps	0ps	750ps	400ps	1.88
b.	500ps	220ps	280ps	0ps	1000ps	500ps	2.00

Note that the " ideal " speed-up from pipelining is now 4 (we removed the MEM stage), and the actual speed-up is about half of that.

4.37.6   For the single cycle design, we can reduce the clock cycle time by 1ps by reducing the latency of any component on the critical path by 1ps (if there is only one critical path). For a pipelined design, we must reduce latencies of all stages that have longer latencies than the target latency. We have:

	Single-cycle	Needed cycle time for pipelined	Cost for Pipelined
a.	0.2 × 1130 = $226	0.8 × 400ps = 320ps	$80 + $30 = $130 (IF and MEM)
b.	0.2 × 2100 = $420	0.8 × 1000ps = 800ps	$200 (MEM)

Note that the cost of improving the pipelined design by 20% is lower. This is because its clock cycle time is already lower, so a 20% improvement represents fewer picoseconds (and fewer dollars in our problem).

Solution 4.38

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

a.	100pJ + 2 × 60pJ + 70pJ = 290pJ
b.	200pJ + 2 × 90pJ + 80pJ = 460pJ

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

a.	100pJ + 2 × 60pJ + 70pJ + 120pJ = 410pJ
b.	200pJ + 2 × 90pJ + 80pJ + 300pJ = 760pJ

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

	Energy before change	Energy saved by change	% Savings
a.	100pJ + 2 × 60pJ + 70pJ + 120pJ = 410pJ	60pJ	14.6%
b.	200pJ + 2 × 90pJ + 80pJ + 300pJ = 760pJ	90pJ	11.8%

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

	Clock cycle time before change	Clock cycle time after change
a.	400ps (I-Mem in IF stage)	500ps (Ctl then Regs in ID stage)
b.	1000ps (D-Mem in MEM stage)	No change (400ps + 220ps < 1000ps).

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

4.38.6

	I-Mem active energy	I-Mem latency	Clock cycle time	Total I-Mem Energy	Idle energy %
a.	100pJ	400ps	400ps	100pJ	0%
b.	200pJ	500ps	1000ps	200pJ + 500ps $\times$ 0.1 $\times$ 200pJ/500ps = 220pJ	20pJ/220pJ = 9.1%

Solution 4.39

4.39.1   The number of instructions executed per second is equal to the number of instructions executed per cycle (IPC, which is 1/CPI) times the number of cycles per second (clock frequency, which is 1/T where T is the clock cycle time). The IPC is he percentage of cycle in which we complete an instruction (and not a stall), and the clock cycle time is the latency of the maximum-latency pipeline stage. We have:

	IPC	Clock cycle time	Clock frequency	Instructions per second
a.	0.85	500ps	2.00 GHz	1.70×10^9
b.	0.70	200ps	5.00 GHz	3.50×10^9

4.39.2 Power is equal to the product of energy per cycle times the clock frequency (cycles per second). The energy per cycle is the total of the energy expenditures in all ?ve stages. We have:

	Clock Frequency	Energy per cycle (in pJ)	Power (W)
a.	2.00 GHz	120 + 60 + 75 + 0.30 × 120 + 0.55 × 20 = 305	0.61
b.	5.00 GHz	150 + 60 + 50 + 0.35 × 150 + 0.50 × 20 = 322.5	1.61

4.39.3 The time that remains in the clock cycle after a circuit completes its work is often called slack. We determine the clock cycle time and then the slack for each pipeline stage:

	Clock cycle time	IF slack	ID slack	EX slack	MEM slack	WB slack
a.	500ps	200ps	100ps	150ps	0ps	400ps
b.	200ps	0ps	50ps	80ps	10ps	60ps

4.39.4 All stages now have latencies equal to the clock cycle time. For each stage, we can compute the factor X for it by dividing the new latency (clock cycle time) by the original latency. We then compute the new per-cycle energy consumption for each stage by dividing its energy by its factor X. Finally, we re-compute the power dissipation:

	X for IF	X for ID	X for EX	X for MEM	X for WB	New Power (W)
a.	500/300	500/400	500/350	500/500	500/100	0.43
b.	200/200	200/150	200/120	200/190	200/140	1.41

4.39.5 This changes the clock cycle time to 1.1 of the original, which changes the factor X for each stage and the clock frequency. After that this problem is solved in the same way as all stages now have latencies equal to the clock cycle time. For each stage, we can compute the factor X for it by dividing the new latency (clock cycle time) by the original latency. We then compute the new per-cycle energy consumption for each stage by dividing its energy by its factor X. Finally, we re-compute the power dissipation:. We get:

	X for IF	X for ID	X for EX	X for MEM	X for WB	New Power (W)
a.	550/300	550/400	550/350	550/500	550/100	0.35
b.	220/200	220/150	220/120	220/190	220/140	1.16

4.39.6   The X factor for each stage is the same as in this changes the clock cycle time to 1.1 of the original, which changes the factor X for each stage and the clock frequency. After that this problem is solved in the same way as all stages now have latencies equal to the clock cycle time. For each stage, we can compute the factor X for it by dividing the new latency (clock cycle time) by the original latency. We then compute the new per-cycle energy consumption for each stage by dividing its energy by its factor X. Finally, we re-compute the power dissipation:. We get:, but this time in our power computation we divide the per-cycle energy of each stage by X^2 instead of x. We get:

	New Power (W)	Old Power (W)	Saved
a.	0.24	0.61	60.7%
b.	0.95	1.61	41.0%

5

Solutions

5.1

5.1.1   4

5.1.2   I, J

5.1.3   A[I][J]

5.1.4   3596 ? 8 ? 800/4 ? 2? 8? 8/4 ? 8000/4

5.1.5   I, J

5.1.6   A(J, I)

5.2

5.2.1

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

5.2.2

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

5.2.3

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

Cache 1 miss rate? 100%

Cache 1 total cycles? 12 ? 25 ? 12 ? 2 ? 324

Cache 2 miss rate? 10/12 ? 83%

Cache 2 total cycles? 10 ? 25 ? 12 ? 3 ? 286

Cache 3 miss rate? 11/12 ? 92%

Cache 3 total cycles? 11 ? 25 ? 12 ? 5 ? 335

Cache 2 provides the best performance.

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

The total cache size can be generalized to

totalsize ? datasize? (validbitsize ? tagsize) ? blocks

totalsize ? 41984

datasize? blocks ? blocksize ? wordsize

wordsize ? 4

tagsize ? 32 ? log2(blocks) ? log2(blocksize) ? log2(wordsize)

validbitsize ? 1

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

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

41984 ?? 64 ? blocks ? 15 ? blocks

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

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

5.2.5   Associative caches are designed to reduce the rate of confl ict misses. As such, a sequence of read requests with the same 12-bit index fi eld but a diff erent tag fi eld will generate many misses. For the cache described above, the sequence 0, 32768, 0, 32768, 0, 32768,   … , would miss on every access, while a 2-way set associate cache with LRU replacement, even one with a signifi cantly smaller overall capacity, would hit on every access after the fi rst two.

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

5.3

5.3.1   8

5.3.2   32

5.3.3   1? (22/8/32) ? 1.086

5.3.4   3

5.3.5   0.25

5.3.6   ? Index, tag, data?

    ? 000001_2, 0001_2, mem[1024] ?

    ? 000001_2, 0011_2, mem[16] ?

    ? 001011_2, 0000_2, mem[176] ?

    ? 001000_2, 0010_2, mem[2176] ?

    ? 001110_2, 0000_2, mem[224] ?

    ? 001010_2, 0000_2, mem[160] ?

5.4

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

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

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

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

The instruction bandwidth is thus (0.0030 ? 64) ? 0.5 ? 0.096 bytes/cycle. The data read bandwidth is thus 0.02 ? (0.13? 0.050) ? 64 ? 0.23 bytes/cycle. Th e total read bandwidth requirement is 0.33 bytes/cycle. Th e data write bandwidth requirement is 0.05 ? 4 ? 0.2 bytes/cycle.

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

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

The instruction bandwidth is (0.0030 ? 64) ? 0.67? 0.13 bytes/cycle.

For the write-through cache, the data read bandwidth is 0.02 ? (0.17 ? 0.067) ? 64 ? 0.22 bytes/cycle. The total read bandwidth is 0.35 bytes/cycle. Th e data write bandwidth is 0.067 ? 4 ? 0.27 bytes/cycle.

For the write-back cache, the data write bandwidth becomes 0.02 ? 0.30 ? (0.17? 0.067) ? 64 ? 0.091 bytes/cycle.

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

5.5

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

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

5.5.3   In this case the miss rate is 0.

5.5.4   AMAT for B ? 8: 0.040? (20 ? 8) ? 6.40

        AMAT for B ? 16: 0.030? (20 ? 16) ? 9.60

        AMAT for B ? 32: 0.020? (20 ? 32) ? 12.80

        AMAT for B ? 64: 0.015? (20 ? 64) ? 19.20

        AMAT for B ? 128: 0.010? (20 ? 128) ? 25.60

        B ? 8 is optimal.

5.5.5   AMAT for B ? 8: 0.040? (24 ? 8) ? 1.28

        AMAT for B ? 16: 0.030? (24 ? 16) ? 1.20

        AMAT for B ? 32: 0.020? (24 ? 32) ? 1.12

        AMAT for B ? 64: 0.015? (24 ? 64) ? 1.32

        AMAT for B ? 128: 0.010? (24 ? 128) ? 1.52

        B ? 32 is optimal.

5.5.6   B? 128

5.6

5.6.1

P1	1.52 GHz
P2	1.11 GHz

5.6.2

P1	6.31 ns	9.56 cycles
P2	5.11 ns	5.68 cycles

5.6.3

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

5.6.4

| 6.50 ns | 9.85 cycles | Worse |

5.6.5   13.04

5.6.6   P1 AMAT ? 0.66 ns ? 0.08 ? 70 ns ? 6.26 ns

P2 AMAT ? 0.90 ns ? 0.06 ? (5.62 ns ? 0.95 ? 70 ns) ? 5.23 ns

For P1 to match P2's performance:

5.23 ? 0.66 ns ? MR ? 70 ns

MR ? 6.5%

5.7

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

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

44	0010 1100	2	6	M	T(1)? 0 T(2)? 11 T(5)? 2 T(7)? 11 T(4)? 5 T(6)? 2	T(1)? 0 T(7)? 0	
186	1011 1010	11	5	M	T(1)? 0 T(2)? 11 T(5)? 2 T(7)? 11 T(4)? 5 T(6)? 2	T(1)? 0 T(7)? 0 T(5)? 11	
253	1111 1101	15	6	M	T(1)? 0 T(2)? 11 T(5)? 2 T(7)? 11 T(4)? 5 T(6)? 2	T(1)? 0 T(7)? 0 T(5)? 11 T(6)? 15	

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

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

5.7.3

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

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

Miss rate ? 9/12 ? 75%

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

5.7.4 L1 only:

.07? 100? 7 ns

CPI ? 7 ns / .5 ns? 14

Direct mapped L2:

.07? (12 ? 0.035? 100) ? 1.1 ns

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

8-way set associated L2:

.07? (28 ? 0.015? 100) ? 2.1 ns

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

Doubled memory access time, L1 only:

.07? 200? 14 ns

CPI ? 14 ns / .5 ns? 28

Doubled memory access time, direct mapped L2:

.07? (12 ? 0.035? 200) ? 1.3 ns

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

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

.07? (28 ? 0.015? 200) ? 2.2 ns

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

Halved memory access time, L1 only:

.07? 50 ? 3.5 ns

CPI ? 3.5 ns / .5 ns? 7

Halved memory access time, direct mapped L2:

.07? (12 ? 0.035? 50) ? 1.0 ns

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

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

.07 ? (28 ? 0.015 ? 50) ? 2.1 ns

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

5.7.5   .07 ? (12 ? 0.035 ? (50 ? 0.013 ? 100)) ? 1.0 ns

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

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

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

5.8

5.8.1

1096 days	26304 hours

5.8.2

0.9990875912%

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

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

5.9

5.9.1   Need to fi nd minimum p such that $2^{p} ?? p ? d ? 1$ and then add one. Th us 9 total bits are needed for SEC/DED.

5.9.2   The (72,64) code described in the chapter requires an overhead of 8/64 ? 12.5% additional bits to tolerate the loss of any single bit within 72 bits, providing a protection rate of 1.4%. Th   e (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%. Th e cost/performance of both codes is as follows:

(72,64) code ??   12.5/1.4 ? 8.9

(136,128) code ??   7.0/0.73 ? 9.6

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

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

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

5.10.1 32 KB

5.10.2 Still 32 KB

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

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

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

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

5.11

5.11.1

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

5.11.2

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

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

5.11.3   Two-way set associative

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

Direct mapped

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

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

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

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

5.11.5   In the two-level approach, the 2^{19} page table entries are divided into 256 segments that are allocated on demand. Each of the second-level tables contain $2^{(19-8)} = 2048$ entries, requiring $2048 \times 4 = 8$ 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 fi rst-level tables would require an additional 5 ? 128 ? 6 bytes? 3840 bytes.

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

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

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

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

5.12

5.12.1 Worst case is $2^{(43 ? 12)}$ entries, requiring $2^{(43 ? 12)}$? 4 bytes ? 2^{33} ? 8 GB.

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

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

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

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

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

5.13

5.13.1 0 hits

5.13.2 1 hit

5.13.3 1 hits or fewer

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

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

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

5.14

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

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

　　　　Native: L; NPT: L ? (L ? 2)

5.14.3 Shadow page table: page fault rate.

　　　　NPT: TLB miss rate.

5.14.4 Shadow page table: 1.03

　　　　NPT: 1.04

5.14.5 Combining multiple page table updates

5.14.6 NPT caching (similar to TLB caching)

5.15

5.15.1 CPI ? 1.5 ? 120/10000 ? (15 ? 175) ? 3.78

　　　　If VMM performance impact doubles　　?? CPI ? 1.5 ? 120/10000 ? (15 ? 350) ? 5.88

　　　　If VMM performance impact halves　　?? CPI ? 1.5 ? 120/10000 ? (15 ? 87.5) ? 2.73

5.15.2 Non-virtualized CPI ? 1.5 ? 30/10000 ? 1100 ? 4.80

　　　　Virtualized CPI   ? 1.5 ? 120/10000 ? (15 ? 175) ? 30/10000 ? (1100 ? 175) ? 7.60

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

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

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

5.15.4   Emulating a diff erent 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. Th is can require many more instructions to be executed to emulate each instruction than was originally necessary in the target ISA. Th is can cause a large performance impact and make it diffi cult 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 ISA has a single instruction that can handle the execution of several of the emulated system 's instructions, then potentially the number of instructions executed can be reduced. This is similar to the case with the recent Intel processors that do micro-op fusion, allowing several instructions to be handled by fewer instructions.

5.16

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

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

5.16.3   Correct solutions should exhibit the following features:

1. The memory read should come before memory writes.

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

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

5.17

5.17.1   There are 6 possible orderings for these instructions.

Ordering 1:

P1	P2
X[0]?? ;	
X[1] ? 3;	
	X[0] ? 5
	X[1] ?? 2;

Results: (5,5)

Ordering 2:

P1	P2
X[0]?? ;	
	X[0] ? 5
X[1] ? 3;	
	X[1] ?? 2;

Results: (5,5)

Ordering 3:

P1	P2
	X[0] ? 5
X[0] ?? ;	
	X[1] ?? 2;
X[1] ? 3;	

Results: (6,3)

Ordering 4:

P1	P2
X[0] ?? ;	
	X[0] ? 5
	X[1] ?? 2;
X[1] ? 3;	

Results: (5,3)

Ordering 5:

P1	P2
	X[0] ? 5
X[0] ?? ;	
X[1] ? 3;	
	X[1] ?? 2;

Results: (6,5)

Ordering 6:

P1	P2
	X[0] ? 5
	X[1] ?? 2;
X[0] ?? ;	
X[1] ? 3;	

(6,3)

If coherency isn't ensured:

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

5.17.2

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

5.17.3 Best case:

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

Worst case:

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

5.17.4 Ordering 1:

P1	P2
A ? 1	
B ? 2	
A ?? 2;	
B?? ;	
	C ? B
	D ? A

Result: (3,3)

Ordering 2:

P1	P2
A ? 1	
B ? 2	
A ?? 2;	
	C ? B
B?? ;	
	D ? A

Result: (2,3)

Ordering 3:

P1	P2
A ? 1	
B ? 2	
	C ? B
A ?? 2;	
B?? ;	
	D ? A

Result: (2,3)

Ordering 4:

P1	P2
A ? 1	
	C ? B
B ? 2	
A ?? 2;	
B?? ;	
	D ? A

Result: (0,3)

Ordering 5:

P1	P2
	C ? B
A ? 1	
B ? 2	
A ?? 2;	
B?? ;	
	D ? A

Result: (0,3)

Ordering 6:

P1	P2
A ? 1	
B ? 2	
A ?? 2;	
	C ? B
	D ? A
B?? ;	

Result: (2,3)

Ordering 7:

P1	P2
A ? 1	
B ? 2	
	C ? B
A ?? 2;	
	D ? A
B?? ;	

Result: (2,3)

Ordering 8:

P1	P2
A ? 1	
	C ? B
B ? 2	
A ?? 2;	
	D ? A
B?? ;	

Result: (0,3)

Ordering 9:

P1	P2
	C ? B
A ? 1	
B ? 2	
A ?? 2;	
	D ? A
B?? ;	

Result: (0,3)

Ordering 10:

P1	P2
A ? 1	
B ? 2	
	C ? B
	D ? A
A ?? 2;	
B?? ;	

Result: (2,1)

Ordering 11:

P1	P2
A ? 1	
	C ? B
B ? 2	
	D ? A
A ?? 2;	
B?? ;	

Result: (0,1)

Ordering 12:

P1	P2
	C ? B
A ? 1	
B ? 2	
	D ? A
A ?? 2;	
B?? ;	

Result: (0,1)

Ordering 13:

P1	P2
A ? 1	
	C ? B
	D ? A
B ? 2	
A ?? 2;	
B?? ;	

Result: (0,1)

Ordering 14:

P1	P2
	C ? B
A ? 1	
	D ? A
B ? 2	
A ?? 2;	
B?? ;	

Result: (0,1)

Ordering 15:

P1	P2
	C ? B
	D ? A
A ? 1	
B ? 2	
A ?? 2;	
B?? ;	

Result: (0,0)

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

Result: (2,0)

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

The allocation policy has little eff ect on the protocol.

5.18

5.18.1   Benchmark A

$AMAT_{private} ? (1/32) ? 5 ? 0.0030 ? 180 ? 0.70$

$AMAT_{shared} ? (1/32) ? 20 ? 0.0012 ? 180 ? 0.84$

Benchmark B

$AMAT_{private} ? (1/32) ? 5 ? 0.0006 ? 180 ? 0.26$

$AMAT_{shared} ? (1/32) ? 20 ? 0.0003 ? 180 ? 0.68$

Private cache is superior for both benchmarks.

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

Benchmark A

$AMAT_{private} ? (1/32) ? 5 ? 0.0030 ? 360 ? 1.24$

$AMAT_{shared} ? (1/32) ? 40 ? 0.0012 ? 180 ? 1.47$

Benchmark B

$AMAT_{private} ? (1/32) ? 5 ? 0.0006 ? 360 ? 0.37$

$AMAT_{shared} ? (1/32) ? 40 ? 0.0003 ? 180 ? 1.30$

Private is still superior for both benchmarks.

5.18.3

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

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

5.18.5   4 times.

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

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

   Caches: more miss status handling registers (MSHR)

   Memory: memory controller to support multiple outstanding memory requests

5.19

5.19.1   srcIP   and refTime   fields. 2 misses per entry.

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

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

Group srcIP , refTime   and status together.

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

Conflict misses do not occur in fully associative caches.

Compulsory (cold) misses are not affected by associativity.

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

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

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

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

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

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

6 Solutions

Solution 6.1

6.1.1

a.	Video Game	Controller —Input, Human Monitor —Output, Human CDROM —Storage, Machine
b.	Handheld GPS	Keypad—Input, Human Display —Output, Human Satellite Interface —Input, Machine Computer Interface —I/O, Machine

6.1.2

a.	Video Game	Controller —0.0038 Mbit/sec Monitor —800– 8000 Mbit/sec CDROM—88– 220 Mbit/sec
b.	Handheld GPS	Keypad—0.0001 Mbit/sec Display —800 Mbit/sec Satellite Interface —10 Mbit/sec Computer Interface —400– 800 Mbit/sec

6.1.3

a.	Video Game	Controller —Operation Rate Monitor —Data Rate CDROM—Data Rate for most applications
b.	Handheld GPS	Keypad—Operation Rate Display —Data Rate Satellite Interface —Data Rate Computer Interface —Data Rate

Solution 6.2

6.2.1

a.	43848
b.	87480

6.2.2

a.	0.996168582375479
b.	0.998628257887517

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

6.2.4   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 speci? c value of MTTR is not signi? cant.

Solution 6.3

6.3.1

a.	15.196 ms
b.	13.2 ms

6.3.2

a.	15.225
b.	13.233

6.3.3   The dominant factor for all disks seems to be the average seek time, although RPM would make a signi? cant contribution as well. Interestingly, by doubling the block size, the RW time changes very little. Thus, block size does not seem to be critical.

Solution 6.4

6.4.1

a.	Yes	A satellite database will process infrequent requests for bulk information. Thus, increasing the sector size will allow more data per read request.
b.	No	This depends substantially on which aspect of the video game is being discussed. However, response time is critical to gaming. Increasing sector size may reduce response time.

6.4.2

a.	Yes	Increasing rotational speed will allow more data to be retrieved faster. For bulk data, this should improve performance.
b.	No	Increasing rotational speed will allow improved performance when retrieving graphical elements from disk.

6.4.3

a.	No	A database system that is collecting data must have exceptionally high availability, or data loss is possible.
b.	Yes	Increasing disk performance in a non-critical application such as this may have bene? ts.

Solution 6.5

6.5.1   There is no penalty for either seek time or for the disk rotating into position to access memory. In effect, if data transfer time remains constant, performance should increase. What is interesting is that disk data transfer rates have always outpaced improvements with disk alternatives. Flash is the ?   rst technology with potential to catch hard disk.

6.5.2

a.	No	Databases are huge and Flash is expensive. The performance gain is not worth the expense.
b.	Yes	Anything that improves performance is of bene? t to gaming.

6.5.3

a.	Maybe	Decreasing download time is highly bene? cial to database downloads. However, the data rate for some satellites may be so low that no gain would result.
b.	Yes	

Solution 6.6

6.6.1   Note that some of the speci?   ed Flash memories are controller limited. This is to convince you to think about the system rather than simply the Flash memory.

a.	28.7 ms
b.	32.5 ms

6.6.2 Note that some of the speci? ed Flash memories are controller limited. This is to convince you to think about the system rather than simply the Flash memory.

a.	14.35 ms
b.	16.25 ms

6.6.3 On initial thought, this may seem unexpected. However, as the Flash memory array grows, delays in propagation through the decode logic and delays propagating decoded addresses to the Flash array account for longer access times.

Solution 6.7

6.7.1

a.	Asynchronous. Mouse inputs are relatively infrequent in comparison to other inputs. The mouse device is electrically distant from the CPU.
b.	Synchronous. The memory controller is electrically close to the CPU and throughput to memory must be high.

6.7.2 For all devices in the table, problems with long, synchronous buses are the same. Speci?cally, long synchronous buses typically use parallel cables that are subject to noise and clock skew. The longer a parallel bus is, the more susceptible it is to environmental noise. Balanced cables can prevent some of these issues, but not without signi? cant expense. Clock skew is also a problem with the clock at the end of a long bus being delayed due to transmission distance or distorted due to noise and transmission issues. If a bus is electrically long, then an asynchronous bus is usually best.

6.7.3 The only real drawback to an asynchronous bus is the time required to transmit bulk data. Usually, asynchronous buses are serial. Thus, for large data sets, transmission can be quite high. If a device is time sensitive, then an asynchronous bus may not be the right choice. There are certainly exceptions to this rule-of-thumb such as FireWire, an asynchronous bus that has excellent timing properties.

Solution 6.8

6.8.1

a.	USB or FireWire due to hot swap capabilities and access to the drive.
b.	USB due to distance from the CPU and low bandwidth requirements. FireWire would not be as appropriate due to its daisy chaining implementation.

6.8.2

Bus Type	Protocol
PCI	Uses a single, parallel data bus with control lines for each device. Individual devices do not have controllers, but send requests and receive commands from the bus controller through their control lines. Although the data bus is shared among all devices, control lines belong to a single device on the bus.
USB	Similar to the PCI bus except that data and control information is communicated serially from the bus controller.
FireWire	Uses a daisy chain approach. A controller exists in each device that generates requests for the device and processes requests from devices after it on the bus. Devices relay requests from other devices along the daisy chain until they reach the main bus controller.
SATA	As the name implies, Serial ATA uses a serial, point-to-point connection between a controller and device. Although both SATA and USB are serial connections, point-to-point implies that unlike USB, data lines are not shared by multiple connections. Like USB and FireWire, SATA devices are hot swappable.

6.8.3

Bus Type	Drawbacks
PCI	The parallel bus use to transmit data limits the length of the bus. Having a ? xed number of control lines limits the number of devices on the bus. The tradeoff is speed. PCI buses are not useful for peripherals that are physically distant from the computer.
USB	Serial communication implies longer communication distances, but the serial nature of the communication limits communication speed. USB buses are useful for peripherals with relatively low data rates that must be physically distant from the computer.
FireWire	Daisy chaining allows adding theoretically unlimited numbers of devices. However, when one device in the daisy chain dies, all devices further along the chain cannot communicate with the controller. The multiplexed nature of communication on FireWire makes it faster than USB.
SATA	The high-speed nature of SATA connections limits the length of the connection between the controller and devices. The distance is longer than PCI, but shorter than FireWire or USB. Because SATA connections are point-to-point, SATA is not as extensible as either USB or FireWire.

Solution 6.9

6.9.1   A polled device is checked by devices that communicate with it. When the devices requires attention or is available, the polling process communicates with it.

a.	No. Signals from the controller must be handed immediately for satisfactory interaction.
b.	Yes

6.9.2   Interrupt-driven communication involves devices raising interrupts when they require attention and the CPU processing those interrupts as appropriate. While polling requires a process to periodically examine the state of a device, interrupts are raised by the device and occur when the device is ready to communicate. When the CPU is ready to communicate with the device, the handler associated with the interrupt runs and then returns control to the main process.

a.	Inputs from the controller generate interrupts handled by the controller driver.
b.	Polling is okay

6.9.3   Basically, each interface is designed in a similar way with memory locations identi? ed for inputs and outputs associated with devices.

a.	The video game controller is an input only device. It has 4 buttons, a joystick, and a rocker. Each button can be in either an on or off position. The joystick generates nine 1 byte values that indicate relative position as a vector from the origin at the center of the control. Finally, the rocker state is expressed as 4 bits, one bit for each of the four directions.
b.	A computer monitor is an output only device that requires memory based on the number of pixels available for output. The monitor I am sitting at now is 2560x1600 pixels. Each pixel requires a memory word to set its color. The amount of memory required suggests why video cards tend to have their own, onboard memory.

6.9.4

a.	The video game controller is an input only device. Thus, a collection of commands should be de?ned that either poll inputs or are called to process interrupts. The commands simply convey the same information as memory mapping, but return values for command invocations.
b.	A computer monitor is an output only device A single command can be implemented that sends an image to be displayed to the interface card. Alternatively, it could send a pointer to the image rather than the image itself.

6.9.5   Absolutely. A graphics card is an excellent example. A memory map can be used to store information that is to be displayed. Then, a command used to actually display the information. Similar techniques would work for other devices from the table.

Solution 6.10

6.10.1   Low-priority interrupts are disabled to prevent them from interrupting the handing of the current interrupt which is higher priority. The status register is saved to assure that any lower priority interrupts that have been detected are handled with the status register is restored following handing of the current interrupt.

6.10.2   Lower numbers have higher interrupt priorities

a.	Power Down: 2	Overheat: 1	Ethernet Controller Data: 3
b.	Overheat: 1	Reboot: 2	Mouse Controller: 3

6.10.3

Power Down Interrupt	Jump to an emergency power down sequence and begin execution
Ethernet Controller Data Interrupt	Save the current program state. Jump to the Ethernet controller code and handle data input. Restore the program state and continue execution
Overheat Interrupt	Jump to an emergency power down sequence and begin execution
Mouse Controller Interrupt	Save the current program state. Jump to the mouse controller code and handle input. Restore the program state and continue execution
Reboot Interrupt	Jump to address 0 and reinitialize the system

6.10.4   If the enable bit of the cause register is not set then interrupts are all disabled and no interrupts will be handled. Zeroing all bits in the mask would have the same affect.

6.10.5   Hardware support for saving and restoring program state prior to interrupt-handling would help substantially. Speci? cally, when an interrupt is handled that does not terminate execution, the running program must return to the point where the interrupt occurred. Handling this in the operating system is certainly feasible, but the only solution requires storing information on a stack or some other dedicated memory area. In some case, registers are dedicated to this task. Providing hardware support removes the burden from the operating system and program state need not be pulled from the CPU and put in memory.

This is essentially the same has handling a function call, except that some interrupts do not allow the interrupted program to resume execution. Like an interrupt, a function must store program state information before jumping to its code. There are sophisticated activation record management protocols and frequently supporting hardware for many CPUs.

6.10.6   Priority interrupts can still be implemented by the interrupt handler in roughly the same manner. Higher priority interrupts are handled ? rst and lower priority interrupts are disabled when a higher priority interrupt is being handled. Even though each interrupt causes a jump to its own vector, the interrupt system implementation must still handle interrupt signals.

Both approaches have roughly the same capabilities.

Solution 6.11

6.11.1   Yes. The CPU initiates the data transfer, but once the data transfer starts, the device and memory communicate directly with no intervention from the CPU.

6.11.2

a.	Yes. If the CPU is processing graphical data that is to be displayed, allowing the graphics card to access that data without going through the CPU can prevent substantial delays.
b.	Yes. If the CPU is processing sound data that is to be output by the sound card in real time, allowing the sound card to access data without going through the CPU can have extensive bene?t.

DMA is useful when individual transactions with the CPU may involve large amounts of data. A frame handled by a graphics card may be huge, but is treated as one display action. Conversely, input from a mouse is tiny.

6.11.3

a.	No. The graphics card does not write back to system memory.
b.	No. The sound card does not write back to system memory.

Basically, any device that writes to memory directly can cause the data in memory to differ from what is stored in cache.

6.11.4   Virtual memory swaps memory pages in and out of physical memory based on locations being addressed. If a page is not in memory when an address associated with it is accessed, the page must be loaded, potentially displacing another page. Virtual memory works because of the principle of locality. Speci?cally, when memory is accessed, the likelihood of the next access being nearby is high. Thus, pulling a page from disk to memory due to a memory access not only retrieves the memory be accessed, but likely the next memory element being access.

Any of the devices listed in the table could cause potential problems if it causes virtual memory to thrash, continuously swapping in and out pages from physical memory. This would happen if the locality principle is violated by the device. Careful design and suf?cient physical memory will almost always solve this problem.

Solution 6.12

6.12.1

a.	Yes.
b.	Yes.

6.12.2

a.	No. Web data is usually small, but requires signi? cant numbers of transactions.
b.	Yes. Sound data is large with relatively infrequent transactions.

6.12.3   See the previous problem for explanations.

a.	Yes.
b.	No.

6.12.4   Polling would be more inappropriate for applications were numbers of transactions handled is a good performance metric. When data throughput dominates numbers of transactions, then polling could potentially be a reasonable approach.

The selection of command-driven or memory-mapped I/O is more dif?   cult. In most situations, a mixture of the two approaches is the most pragmatic approach. Speci?cally, use commands to handle interactions and memory to exchange data. For transaction dominated I/O, command-driven I/O will likely be suf?   cient.

Solution 6.13

6.13.1

a.	Large numbers of small, concurrent transactions
b.	Large, concurrent data reads and writes

6.13.2   Standard benchmarks help when trying to compare and contrast different systems. Ranking systems with benchmarks is generally not useful. However, understanding tradeoffs certainly is.

6.13.3   It does not make much sense to evaluate an I/O system outside the system where it will be used. Although benchmarks help simulate the environment of a system, nothing replaces live data in a live system.

CPUs are particularly dif?   cult to evaluate outside of the system where they are used. Again, benchmarks can help with this, but frequently Amdahl   's Law makes spending resources on improving CPU speed have diminishing returns.

Solution 6.14

6.14.1   Striping forces I/O to occur on multiple disks concurrently rather than on a single disk.

a.	No. The bottleneck in such systems is network throughput, not disk I/O
b.	Yes. Sound editing requires access to large amounts of data in real time.

6.14.2   The MTBF is calculated as MTTF + MTTR, with MTTF as the dominating factor. For the RAID 1 system with redundancy to fail, both disks must fail. The probability of both disks failing is the product of a single disk failing. The result is a substantially increased MTBF.

In all applications, decreasing the likelihood of data loss is good. However, online database and video services are particularly sensitive to resource availability. When such systems are of?ne, revenue loss is immediate and customers lose con?dence in the service.

6.14.3   RAID 1 maintains two complete copies of a dataset while RAID 3 maintains error correction data only. The tradeoff is storage cost. RAID 1 requires 2 times the actual storage capacity while RAID 3 requires substantially less. This must be viewed both in terms of the cost of disks, but also power and other resources required to keep the disk array running.

In the previous applications, large online services like database and video services would de? nitely bene? t from RAID 3. Video and sound editing may also bene?   t from RAID 3, but these applications are not as sensitive to availability issues as online services.

Solution 6.15

6.15.1

a.	513C
b.	8404

6.15.2

a.	BB83
b.	FC4C

6.15.3   RAID 4 is more ef? cient because it requires fewer reads to generate the next parity word value. Speci?cally, RAID 3 accesses every disk for every data write no matter which disk is being written to. For smaller writes where data is located on a single disk, RAID 4 will be more ef? cient.

RAID 3 has no inherent advantages to RAID 4.

6.15.4   RAID 5 distributes parity blocks throughout the disk array rather than on a single disk. This eliminates the parity disk as a bottleneck during disk access. For applications with high numbers of concurrent reads and writes, RAID 5 will be more ef?cient. For lower volume, RAID 5 will not signi? cantly outperform RAID 4.

6.15.5   As the number of disks grows by 1, the number of accesses required to calculate a parity word in RAID 3 also grows by 1. In contrast, RAID 4 and 5 continue to access only existing values of data being stored. Thus, as the number of disks grows, RAID 3 performance will continue to degrade while RAID 4 and 5 will remain constant.

There is no performance advantage for RAID 4 or 5 over RAID three for small numbers of disks. For 2 disks, there is no difference.

Solution 6.16

6.16.1

a.	13333
b.	26667

6.16.2

	16 Disks		8 Disks		4 Disks		2 Disks	
	IOPS	Bottleneck?	IOPS	Bottleneck?	IOPS	Bottleneck?	IOPS	Bottleneck?
a.	14000	No	7000	Yes	3500	Yes	1750	Yes
b.	28000	No	14000	No	7000	Yes	3500	Yes

6.16.3

	PCI Bus		DIMM		Front Side Bus	
	IOPS	Bottleneck?	IOPS	Bottleneck?	IOPS	Bottleneck?
a.	15625	No	41687.5	NO	82812.5	No
b.	31250	No	83375	No	165625	No

6.16.4   The assumptions made in approximating I/O performance are extensive. From the approximation of I/O commands generated by the executing system through sequential and random I/O events handled by disks, the approximations are extensive. By benchmarking in a full system, or executing actual application an engineer can see actual numbers that are far more accurate than approximate calculations.

Solution 6.17

6.17.1   Runtime characteristics vary substantially from application to application. All three applications perform some kind of transaction processing, but

those transactions may be different in nature. A Web server processes numerous transactions typically involving small amounts of data. Thus, transaction through-put is critical. A database server is similar, but the data transferred may be much larger. A bioinformatics data server will deal with huge data sets where transactions processed is not nearly as critical as data throughput.

When identifying the runtime characteristics of the application, you are implicitly identifying characteristics for evaluation. For a web server, transactions per second is a critical metric. For the bioinformatics data server, data throughput is critical. For a database server, you will want to balance both criteria.

6.17.2   It is relatively easy to use online resources to identify potential servers. You may also ?nd advertisements in periodicals from your professional societies or trade journals. You should be able to identify one or more candidates using the cri-teria identi? ed in 6.17.1. If your reasons for selecting the server don   ' t fo the criteria in 6.17.1, something is not right.

6.17.3   In problem 6.16, we used characteristics of a Sun Fire x4150 to attempt to predict its performance. You can use the same data and characteristics here. Remember that the Sun Fire x4150 has multiple con?   gurations. You should con-sider this when you perform your evaluation.

Find similar measurements for the server that you have selected. Most of this data should be available online. If not, contact the company providing the server and see if such data is available.

It   '   s a reasonably simple task to use a spreadsheet to evaluate numerous con?gura-tions and systems simultaneously. If you design your spreadsheet carefully, you can simply enter a table of data and make comparisons quickly. This is exactly what you will do in industry when evaluating systems.

6.17.4   Although analytic analysis is useful when comparing systems, nothing beats hands-on evaluation. There are a number of test suites available that will serve your needs here. Virtually all of them will be available online. Look for benchmarks that generate transactions for the web server, generate large data transfers for the bioinformatics server, and a combination of the two for the database server.

Solution 6.18

6.18.1

a.	8.76
b.	7.008

6.18.2

	7 years	10 years
a.	31.536	227.76
b.	21.024	151.84

6.18.3   Average failure rates of the drives with longer longevity for 7 and 10 years are:

	7 years	10 years
a.	12.264	36.792
b.	8.176	24.528

It is not surprising that with failure rates starting to double 3 years later, we have to replace far fewer disks in the second situation than the ? rst. The ratio of the number of drives replaced in the ?   rst scenario to the number replaced in the second should give us the multiple that we want:

	7 years	10 years
a.	2.57	6.19
b.	2.57	6.19

Solution 6.19

6.19.1   In all cases, no. The objective of the customer is not known. Thus, improving any performance metric by nearly doubling the cost may or may not have an price impact on the company.

6.19.2   As a search engine provider paid by ad hits, throughput is critical. Most HTTP traf? c is small, so the network is not as great a bottleneck as it would be for large data transfers. RAID 0 may be an effective solution. However, RAID 1 will almost certainly not be an effective solution. Increased availability makes our product more attractive, but a 1.6 cost multiple is most likely too high.

RAID 0 is going to increase throughput by 70%, meaning the potential exists to serve 1.7 times as many ads. The cost of this gain is 0.6 of the original price. 1.7 times as many ads for 1.6 times the original cost may justify the upgrade cost.

6.19.3   This problem is not as simple as it would seem at ?   rst glance. As an online backup provider, availability is critical. Thus, using RAID 1 where failure

rate decreases for a 1.6 times cost increase might be worthwhile. However, online backup is more appealing when services are provided quickly making RAID 0 appealing. Remember Amdahl ' s law. Will increasing throughput in the disk array for long data reads and writes result in performance improvements for the system? The network will be our throughput bottleneck, not disk access. RAID 0 will not help much.

RAID 1 has more potential for increased revenue by making the disk array available more. For our original con? guration, we are losing between 12 and 19 disks per 1000 to 1500 every 7 years. If the system lifetime is 7 years, the RAID 1 upgrade will almost certainly not pay for itself even though it addresses the most critical property of our system. Over 10 years, we lose between 30 and 50 drives. If repair times are small, then even over a 10 year span the RAID 1 solution will not be cost effective.

Solution 6.20

6.20.1   The approach to solving this problem is relatively simple once parameters of a bioinformatics simulation are understood. Simulations tend to run days or months. Thus, losing simulation data or having a system failure during simulation are catastrophic events. Availability is therefore a critical evaluation parameter. Additionally, the disk array will be accessed by 1000 parallel processors. Throughput will be a major concern.

The primary role of the power constraint in this problem is to prevent simply maximizing all parameters in the disk array. Adding additional disks and controllers without justi? cation will increase power consumption unnecessarily.

6.20.2   Remember that your system must provide both backup and archiving. Thus, you will need multiple copies of your data and may be required to move those copies offsite. This makes none of the solutions optimal.

RAID or a second backup array provides high speed backup, but does not provide archival capabilities. Magnetic tape allows archiving, but can be exceptionally slow when comparing to disk backups. Online backup automatically achieves archiving, but can be even slower than disks.

6.20.3   Your benchmarks must evaluate backup throughput. Most other parameters that govern selection of a system are relatively well understood and cost being the primary issues to be evaluated.

7 Solutions

Solution 7.1

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 identi? ed.

7.1.1   Any reasonable answer is correct here.

7.1.2   Any reasonable answer is correct here.

7.1.3   Any reasonable answer is correct here.

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

Solution 7.2

7.2.1   While binary search has very good serial performance, it is dif?cult to parallelize without modifying the code. So part A asks to compute the speed-up factor, but increasing X beyond 2 or 3 should have no bene? ts. 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 speed-up. The answer should include a graph, showing that no speed-up is obtained after the values of 1, 2 or 3 (this value depends somewhat on the assumption made) for Y.

7.2.2   In this question, we suggest that we can increase the number of cores to each the number of array elements. Again, given the current code, we really cannot obtain any bene? t 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 speed-up (Y times speed-up), and the comparison can be completed in the amount of time to perform a single comparison.

This problem illustrates that some computations can be done in parallel if serial code is restructured. But more impor   tantly, 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.

Solution 7.3

7.3.1   This is a straightforward computation. The ? rst instruction is executed once, and the loop body is executed 998 times.

Version 1   — 17,965 cycles

Version 2   — 22,955 cycles

Version 3   — 20,959 cycles

7.3.2   Array elements D[j] and D[j - 1] will have loop carried dependencies. These will f3 in the current iteration and f1 in the next iteration.

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

```
        DADDIU r2, r0, 996
        L.D f1,    – 16(r1)
        L.D f2,    – 8(r1)

loop:

        ADD.D f3, f2, f1
        ADD.D f4, f3, f2
        Send (2, f3)
        Send (2, f4)
        S.D f3, 0(r1)
        S.D f4, 8(r1)
        Receive(f5)
        ADD.D f6,f5,f4
        ADD.D f1,f6,f5
        Send (2, f6)
        Send (2, f1)
        S.D. f5, 16(r1)
        S.D  f6, 24(r1)
        S.D  f1  32(r1)
        Receive(f2)
```

```
        S.D  f2  40(r1)
        DADDIU r1, r1, 48
        BNE r1, r2, loop

        ADD.D f3, f2, f1
        ADD. D f4, f3, f2
        ADD.D f6, f5, f4
        S.D f3, 0(r1)
        S.D f4, 8(r1)
        S.D f5, 16(r1)
```

The code on node 2 would look something like this:

```
        DADDIU r3, r0, 0
```

loop:

```
        Receive (f7)
        Receive (f8)
        ADD.D f9, f8, f7
        Send(1, f9)
        Receive (f7)
        Receive (f8)
        ADD.D f9, f8, f7
        Send(1, f9)
        Receive (f7)
        Receive (f8)
        ADD.D f9, f8, f7
        Send(1, f9)
        Receive (f7)
        Receive (f8)
        ADD.D f9, f8, f7
        Send(1, f9)
        DADDIU r3, r3, 1
        BNE r3, 83, loop
```

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

7.3.4   The loop network would need to respond within a single cycle to obtain a speed-up. This illustrates why using distributed message passing is dif? cult when loops contain loop-carried dependencies.

Solution 7.4

7.4.1   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 speed-up when the number of cores is small. We 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 $1 + 2 + 4 + 8 + 16 + \ldots . \log(m)$ processors to obtain speed-up.

7.4.2   In this question, $\log_2(m)$ is the largest value of Y for which we can obtain any speed-up 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.

Solution 7.5

7.5.1   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.

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

Preheat Oven

Mix incredients in bowl A for Cake 1

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

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

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

Finish baking Cake 3. Empty cake pan A.

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

7.5.3   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.

7.5.4   The loop computation is equivalent to the steps involved to make one cake. Given that we have multiple processors (or ovens and cooks), we can execute instructions (or cook multiple cakes) in parallel. The instructions in the loop (or cooking steps) may have some dependencies on prior instructions (or cooking steps) in the loop body (cooking a single cake). Data-level parallelism occurs when loop iterations are independent (i.e., no loop carried dependencies). Task-level parallelism includes any instructions that can be computed on parallel execution units, are similar to the independent operations involved in making multiple cakes.

Solution 7.6

7.6.1   This problem presents an   " embarrassingly parallel   "   computation and asks the student to ?nd the speed-up obtained on a 4-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 a element until two products are available). So in this question, the speed-up should be very close to 4.

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

7.6.3   In this question, we are asked how to ? x 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 processor the matrix index that is computed (i, j) and (i + 1, j) on the same core. This will eliminate false sharing.

Solution 7.7

7.7.1

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

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

Solution 7.8

7.8.1   1 byte $\times$ C entries = number of bytes consumed in the cache for maintaining coherence.

7.8.2   P bytes/entry $\times$ S/T = number of bytes needed to store coherency information in each directory on a single node.

Solution 7.9

7.9.1   There are a number of correct answers since the answer depends upon the write protocol and the cache coherency protocol chosen. First, the write will generate a read from memory of the L2 cache line, and then the line is written to the L1 cache. Any data that was   " dirty "   in L2 that was replaced is written back to memory. The data updated in the block is updated in L1 and L2 (assuming L1 is updated on a write miss). The status of the line is set to   . Speci?c to the " dirty " coherency protocol assumed, on the ?rst read from another node, a cache-to-cache transfer takes place of the entire dirty cache line. Depending on the cache coherency protocol used, the status of the line will be changed (in our answer it will become " shared " in both caches). The other two reads can be serviced from any of the caches on the two nodes with the updated data. The accesses for the other three writes are handled exactly the same way. The key concept here is that all nodes are interrogated on all reads to maintain coherency, and all must respond to service the read miss.

7.9.2   For a directory-based mechanism, since the address space of memory is divided up on a node-by-node basis, only the directory responsible for the address requested needs to be interrogated. The directory controller will then initiate the cache-to-cache transfer, but will not need to bother the L2 caches on the nodes where the line is not present. All state updates are handled locally at the directory. For the last two reads, again the single directory is interrogated and the directory controller initiates the cache-to-cache transfer. But only the two nodes participating in the transfer are involved. This increases the L2 bandwidth since only the minimum number of cache accesses/interrogations are involved in the transaction.

7.9.3   The answer to this question is similar, though there are subtle differences. For the cache-based block status case, all coherency traf? c is managed at the L2 level between CPUs, so this scenario should not change except that reads by the 3 local cores should not generate any coherence messages outside of the CPU. For the directory case, all accesses need to interrogate the directory and the directory controller will initiate cache-to-cache transfers. Again, the number of accesses is greatly reduced using the directory approach.

7.9.4   This is a case of how false sharing can bring a system to its knees. Assuming an invalidate on write policy, for writes on the same CPU, the L1 dirty copy from the ?rst write will be invalidated on the second write, and this same pattern will occur on the third and fourth write. When writes are done on another CPU, then coherence management moves to the L2, and the L2 copy on the ?rst CPU is invalidated. The local write activity is the same as for the ? rst CPU. This repeats for the last two CPUs. Of course, this assumes that the order of the writes is in numerical order, with the group of 4 writes being performed on the same CPU on each core. If we instead assume that consecutive writes are performed by different CPUs each time, then invalidates will take place at the L2 cache level on each write.

Solution 7.10

This question looks at the impact of handling a second memory access when one is pending, given the fact that one is pending.

7.10.1   We will encounter a 25 cycle stall every 150 cycles

7.10.2   We will encounter a 50 cycle stall every 150 cycles

7.10.3   No impact

Solution 7.11

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

7.11.2 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 ?nishes 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 dif? culty is to ? rst 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 de? ned by being closest to the head of the queue. This provides both deadlock prevention and fairness.

7.11.3 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.

7.11.4 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.

Solution 7.12

7.12.1

Core 1	Core 2
A1, A3	B1, B3
A1	B2
A3	B4
A4	

7.12.2

FU1	FU2
A1	A3
A1	
B1	B3
B2	
A2	
A4	
B4	

7.12.3

FU1	FUl2
A1	B1
A1	B2
A2	B3
A3	B4
A4	

Solution 7.13

This is an open-ended question.

Solution 7.14

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

7.14.2   Since this program is highly data parallel and there are no data depen-dencies, a 8X speed-up should be observed. In terms of instructions, the SIMD machine should have fewer instructions (though this will depend upon the SIMD extensions).

Solution 7.15

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.

Solution 7.16

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

Solution 7.17

This is an open-ended programming assignment. The code should be tested for correctness.
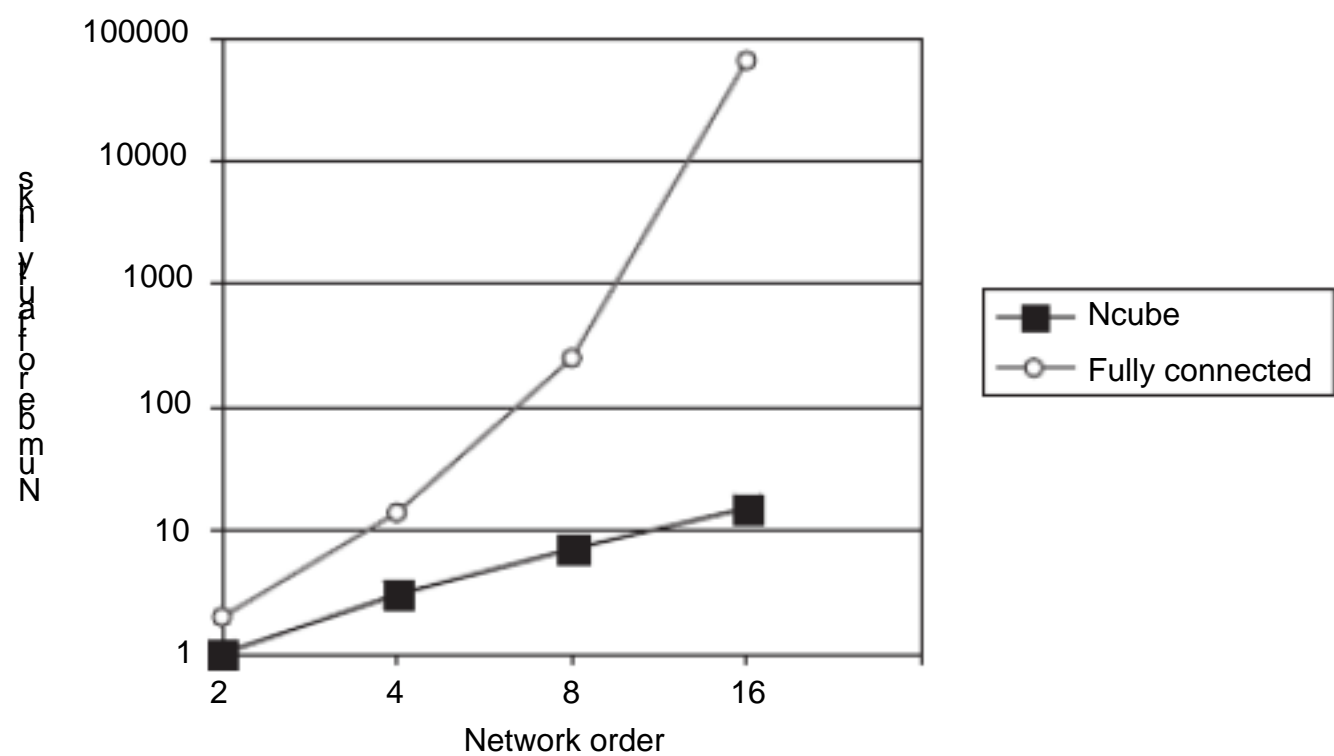
Solution 7.18

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 stu dents become aware of these technologies. The actual bandwidth and la tency values should be available right off the company websites, and will change as the technology evolves.

Solution 7.19

7.19.1 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.

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

Solution 7.20

7.20.1 Major differences between these suites include:

Whetstone — designed for floating point performance speci? cally

PARSEC— these workloads are focused on multithreaded programs

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

Solution 7.21

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

7.21.2   The storage space should be equal to (R + R) times the size of a single-precision ?oating point number + (m + 1) times the size of the index, where R is the number of non-zero elements and m is the number of rows. We will assume each ?oating-point number is 4 bytes, and each index is a short unsigned integer that is 2 bytes.

For Matrix X this equals 62 bytes.

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

7.21.4   There are a number of more ef? cient formats, but their impact should be marginal for the small matrices used in this problem.

Solution 7.22

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

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

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

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

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

7.22.2   The point of the problem is to show that it is dif? cult to perform opera-tion on individual vector elements when utilizing a vector processor. What might be a nice instruction to add would be a vector comparison that would allow for us to compare two vectors and produce scalar value of the number of elements where one vector was larger the other. This would reduce the computation to a single

instruction for the comparison of 8 FP number pairs, and then an integer compu-
tation for summing up all of these values.

Solution 7.23

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 com   puted by the
maximum rate) and the queue time.

7.23.1   So for a max transaction processing rate of 5000/sec, and we have 4 cores
contributing, we would see an average latency of .8 ms if there was no queuing tak-
ing 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

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

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