1 Solutions

Solution 1.1

1.1.1 Computer used to run large problems and usually accessed via a network: (3) servers

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

1.1.3 A class of computers composed of hundred to thousand processors and terabytes of memory and having the highest performance and cost: (5) supercomputers

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) data centers

1.1.8 Microprocessors containing several processors in the same chip: (10) multicore processors

1.1.9 Desktop computer without a screen or keyboard usually accessed via a network: (4) low-end servers

1.1.10 A computer used to running one predetermined application or collection of software: (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 scientific 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 × 3 colors = 24 bits/pixel = 3 bytes/pixel.

AQ 1

a.	Configuration 1: 640 × 480 pixels = 179,200 pixels => 179,200 × 3 = 537,600 bytes/frame Configuration 2: 1280 × 1024 pixels = 1,310,720 pixels => 1,310,720 × 3 = 3,932,160 bytes/frame
b.	Configuration 1: 1024 × 768 pixels = 786,432 pixels => 786,432 × 3 = 2,359,296 bytes/frame Configuration 2: 2560 × 1600 pixels = 4,096,000 pixels => 4,096,000 × 3 = 12,288,000 bytes/frame

1.2.2 No. frames = integer part of (Capacity of main memory/bytes per frame)

a.	Configuration 1: Main memory: 2 GB = 2000 Mbytes. Frame: 537.600 Mbytes => No. frames = 3 Configuration 2: Main memory: 4 GB = 4000 Mbytes. Frame: 3,932.160 Mbytes => No. frames = 1
b.	Configuration 1: Main memory: 2 GB = 2000 Mbytes. Frame: 2,359.296 Mbytes => No. frames = 0 Configuration 2: Main memory: 4 GB = 4000 Mbytes. Frame: 12,288 Mbytes => No. frames = 0

1.2.3 File size: 256 Kbytes = 0.256 Mbytes.

Same solution for a) and b)

Configuration 1: Network speed: 100 Mbit/sec = 12.5 Mbytes/sec. Time = 0.256/12.5 = 20.48 ms
Configuration 2: Network speed: 1 Gbit/sec = 125 Mbytes/sec. Time = 0.256/125 = 2.048 ms

1.2.4

a.	2 microseconds from cache $\Rightarrow$ 20 microseconds from DRAM.
b.	2 microseconds from cache $\Rightarrow$ 20 microseconds from DRAM.

AQ 2

1.2.5

a.	2 microseconds from cache $\Rightarrow$ 2 ms from Flash memory.
b.	2 microseconds from cache $\Rightarrow$ 4.28 ms from Flash memory.

1.2.5

a.	2 microseconds from cache $\Rightarrow$ 2 s from magnetic disk.
b.	2 microseconds from cache $\Rightarrow$ 5.7 s from magnetic disk.

Solution 1.3

1.3.1 P2 has the highest performance.

Instr/sec = f/CPI

a.	performance of P1 (instructions/sec) = $3 \times 10^9/1.5 = 2 \times 10^9$ performance of P2 (instructions/sec) = $2.5 \times 10^9/1.0 = 2.5 \times 10^9$ performance of P3 (instructions/sec) = $4 \times 10^9/2.2 = 1.8 \times 10^9$
b.	performance of P1 (instructions/sec) = $2 \times 10^9/1.2 = 1.66 \times 10^9$ performance of P2 (instructions/sec) = $3 \times 10^9/0.8 = 3.75 \times 10^9$ performance of P3 (instructions/sec) = $4 \times 10^9/2 = 2 \times 10^9$

1.3.2 No. cycles = time × clock rate

time = (No. Instr × CPI)/clock rate, then No. instructions = No. cycles/CPI

a.	cycles(P1) = $10 \times 3 \times 10^9 = 30 \times 10^9$ s cycles(P2) = $10 \times 2.5 \times 10^9 = 25 \times 10^9$ s cycles(P3) = $10 \times 4 \times 10^9 = 40 \times 10^9$ s No. instructions(P1) = $30 \times 10^9/1.5 = 20 \times 10^9$ No. instructions(P2) = $25 \times 10^9/1 = 25 \times 10^9$ No. instructions(P3) = $40 \times 10^9/2.2 = 18.18 \times 10^9$
b.	cycles(P1) = $10 \times 2 \times 10^9 = 20 \times 10^9$ s cycles(P2) = $10 \times 3 \times 10^9 = 30 \times 10^9$ s cycles(P3) = $10 \times 4 \times 10^9 = 40 \times 10^9$ s No. instructions(P1) = $20 \times 10^9/1.2 = 16.66 \times 10^9$ No. instructions(P2) = $30 \times 10^9/0.8 = 37.5 \times 10^9$ No. instructions(P3) = $40 \times 10^9/2 = 20 \times 10^9$

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

AQ 3

a.	$CPI_{new} = CPI_{old} \times 1.2$, then CPI(P1) = 1.8, CPI(P2) = 1.2, CPI(P3) = 2.6 f = No. Instr × CPI/time, then f(P1) = $20 \times 10^9 \times 1.8 / 7 = 5.14$ GHz f(P2) = $25 \times 10^9 \times 1.2 / 7 = 4.28$ GHz f(P1) = $18.18 \times 10^9 \times 2.6 / 7 = 6.75$ GHz
b.	$CPI_{new} = CPI_{old} \times 1.2$, then CPI(P1) = 1.44, CPI(P2) = 0.96, CPI(P3) = 2.4 f = No. Instr × CPI/time, then f(P1) = $16.66 \times 10^9 \times 1.44/7 = 3.42$ GHz f(P2) = $37.5 \times 10^9 \times 0.96/7 = 5.14$ GHz f(P1) = $20 \times 10^9 \times 2.4/7 = 6.85$ GHz

1.3.4 IPC = 1/CPI = No. instr/(time × clock rate)

a.	IPC(P1) = 0.95 IPC(P2) = 1.2 IPC(P3) = 2.5
b.	IPC(P1) = 2 IPC(P2) = 1.25 IPC(P3) = 0.89

1.3.5

a.	$Time_{new}/Time_{old} = 7/10 = 0.7$. So $f_{new} = f_{old}/0.7 = 2.5$ GHz/0.7 = 3.57 GHz.
b.	$Time_{new}/Time_{old} = 5/8 = 0.625$. So $f_{new} = f_{old}/0.625 = 4.8$ GHz.

1.3.6

a.	$Time_{new}/Time_{old} = 9/10 = 0.9$. Then $Instructions_{new} = Instructions_{old} \times 0.9 = 30 \times 10^9 \times 0.9 = 27 \times 10^9$.
b.	$Time_{new}/Time_{old} = 7/8 = 0.875$. Then $Instructions_{new} = Instructions_{old} \times 0.875 = 26.25 \times 10^9$.

Solution 1.4

1.4.1

Class A: 10^5 instr.
Class B: 2×10^5 instr.
Class C: 5×10^5 instr.
Class D: 2×10^5 instr.

Time = No. instr × CPI/clock rate

a.	Total time P1 = $(10^5 + 2 \times 10^5 \times 2 + 5 \times 10^5 \times 3 + 2 \times 10^5 \times 3)/(2.5 \times 10^9) = 10.4 \times 10^{-4}$ s Total time P2 = $(10^5 \times 2 + 2 \times 10^5 \times 2 + 5 \times 10^5 \times 2 + 2 \times 10^5 \times 2)/(3 \times 10^9) = 6.66 \times 10^{-4}$ s
b.	Total time P1 = $(10^5 \times 2 + 2 \times 10^5 \times 1.5 + 5 \times 10^5 \times 2 + 2 \times 10^5)/(2.5 \times 10^9) = 6.8 \times 10^{-4}$ s Total time P2 = $(10^5 + 2 \times 10^5 \times 2 + 5 \times 10^5 + 2 \times 10^5)/(3 \times 10^9) = 4 \times 10^{-4}$ s

1.4.2 CPI = time × clock rate/No. instr

a.	CPI (P1) = $10.4 \times 10^{-4} \times 2.5 \times 10^9/10^6 = 2.6$ CPI (P2) = $6.66 \times 10^{-4} \times 3 \times 10^9/10^6 = 2.0$
b.	CPI (P1) = $6.8 \times 10^{-4} \times 2.5 \times 10^9/10^6 = 1.7$ CPI (P2) = $4 \times 10^{-4} \times 3 \times 10^9/10^6 = 1.2$

1.4.3

a.	clock cycles (P1) = $10^5 \times 1 + 2 \times 10^5 \times 2 + 5 \times 10^5 \times 3 + 2 \times 10^5 \times 3 = 26 \times 10^5$ clock cycles (P2) = $10^5 \times 2 + 2 \times 10^5 \times 2 + 5 \times 10^5 \times 2 + 2 \times 10^5 \times 2 = 20 \times 10^5$
b.	clock cycles (P1) = 17×10^5 clock cycles (P2) = 12×10^5

1.4.4

a.	$(650 \times 1 + 100 \times 5 + 600 \times 5 + 50 \times 2) \times 0.5 \times 10^{-9} = 2{,}125$ ns
b.	$(750 \times 1 + 250 \times 5 + 500 \times 5 + 500 \times 2) \times 0.5 \times 10^{-9} = 2{,}750$ ns

1.4.5 CPI = time × clock rate/No. instr

a.	CPI = $2{,}125 \times 10^{-9} \times 2 \times 10^9/1{,}400 = 3.03$
b.	CPI = $2{,}750 \times 10^{-9} \times 2 \times 10^9/2{,}000 = 2.75$

1.4.6

a.	Time = $(650 \times 1 + 100 \times 5 + 300 \times 5 + 50 \times 2) \times 0.5 \times 10^{-9}$ = 1,375 ns Speedup = 2,125 ns/1,375 ns = 1.54 CPI = $1{,}375 \times 10^{-9} \times 2 \times 10^{9}/1{,}100$ = 2.5
b.	Time = $(750 \times 1 + 250 \times 5 + 250 \times 5 + 500 \times 2) \times 0.5 \times 10^{-9}$ = 2,125 ns Speedup = 2,750 ns/2,125 ns = 1.29 CPI = $2{,}125 \times 10^{-9} \times 2 \times 10^{9}/1{,}750$ = 2.43

Solution 1.5

1.5.1

a.	P1: 2×10^9 inst/sec, P2: 2×10^9 inst/sec
b.	P1: 2×10^9 inst/sec, P2: 3×10^9 inst/sec

1.5.2

a.	T(P2)/T(P1) = 4/7;	P2 is 1.75 times faster than P1
b.	T(P2)/T(P1)= 4.66/5;	P2 is 1.07 times faster than P1

1.5.3

a.	T(P2)/T(P1) = 4.5/8;	P2 is 1.77 times faster than P1
b.	T(P2)/T(P1) = 5.33/5.5;	P2 is 1.03 times faster than P1

1.5.4

a.	2.91 μs
b.	2.50 μs

1.5.5

a.	0.78 μs
b.	0.90 μs

1.5.6

a.	T = 0.68μs => 1.14 times faster
b.	T = 0.75μs => 1.20 times faster

Solution 1.6

1.6.1 CPI = $T_{exec} \times f$/No. Instr

	Compiler A CPI	Compiler B CPI
a.	1.8	1.5
b.	1.1	1.25

1.6.2 f_A/f_B = (No. Instr(A) × CPI(A))/(No. Instr(B) × CPI(B))

a.	$f_A/f_B = 1$
b.	$f_A/f_B = 0.73$

1.6.3

	Speedup vs. Compiler A	Speedup vs. Compiler B
a.	$T_{new}/T_A = 0.36$	$T_{new}/T_B = 0.36$
b.	$T_{new}/T_A = 0.6$	$T_{new}/T_B = 0.44$

1.6.4

	P1 Peak	P2 Peak
a.	4×10^9 Inst/s	2×10^9 Inst/s
b.	4×10^9 Inst/s	3×10^9 Inst/s

1.6.5 Speedup, P1 versus P2:

a.	$T_1/T_2 = 1.9$
b.	$T_1/T_2 = 1.5$

1.6.6

a.	4.37 GHz
b.	6 GHz

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: $2.667 \times 10^9/12.5 \times 10^6 = 213.36$
Power: 95 W/3.3 W = 28.78

1.7.4 $C = P/V^2 \times \text{clock rate}$

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

a.	$C_2/C_1 = 0.9 \times 1.75^2 \times 1.5 \times 10^9/(1.2^2 \times 2 \times 10^9) = 1.43$
b.	$C_2/C_1 = 0.9 \times 1.1^2 \times 3 \times 10^9/(0.8^2 \times 4 \times 10^9) = 1.27$

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

a.	$\text{Power}_2/\text{Power}_1 = 0.62$ => Reduction of 38%
b.	$\text{Power}_2/\text{Power}_1 = 0.7$ => Reduction of 30%

1.8.3

a.	$\text{Power}_2 = V_2^2 \times 2 \times 10^9 \times 0.8 \times C_1 = 0.6 \times \text{Power}_1$ $\text{Power}_1 = 1.75^2 \times 1.5 \times 10^9 \times C_1$ $V_2^2 \times 2 \times 10^9 \times 0.8 \times C_1 = 0.6 \times 1.75^2 \times 1.5 \times 10^9 \times C_1$ $V_2 = ((0.6 \times 1.75^2 \times 1.5)/(2 \times 0.8))^{1/2} = 1.31$ V
b.	$\text{Power}_2 = V_2^2 \times 4 \times 10^9 \times 0.8 \times C_1 = 0.6 \times \text{Power}_1$ $\text{Power}_1 = 1.1^2 \times 3 \times 10^9 \times C_1$ $V_2^2 \times 4 \times 10^9 \times 0.8 \times C_1 = 0.6 \times 1.1^2 \times 3 \times 10^9 \times C_1$ $V_2 = ((0.6 \times 1.1^2 \times 3)/(4 \times 0.8))^{1/2} = 0.825$ V

1.8.4

a.	$\text{Power}_{new} = 1 \times C_{old} \times V^2_{old}/(2^{1/2})^2 \times \text{clock rate} \times 1.15$. Thus, $\text{Power}_{new} = 0.575\ \text{Power}_{old}$
b.	$\text{Power}_{new} = 1 \times C_{old} \times V^2_{old}/(2^{1/4})^2 \times \text{clock rate} \times 1.2$. Thus, $\text{Power}_{new} = 0.848\ \text{Power}_{old}$

1.8.5

a.	$1/2^{1/2} = 0.7$
b.	$1/2^{1/4} = 0.8$

1.8.6

a.	Voltage = $1.1 \times 1/2^{1/2} = 0.77$ V. Clock rate = $2.667 \times 1.15 = 3.067$ GHz.
b.	Voltage = $1.1 \times 1/2^{1/4} = 0.92$ V. Clock rate = $2.667 \times 1.2 = 3.2$ GHz.

Solution 1.9

1.9.1

a.	$10/60 \times 100 = 16.6\%$
b.	$60/150 \times 100 = 40\%$

1.9.2

$$P_{total\_new} = 0.9\ P_{total\_old}$$
$$P_{static\_new}/P_{static\_old} = V_{new}/V_{old}$$

a.	1.08 V
b.	0.81 V

1.9.3

a.	$\text{Power}_{st}/\text{Power}_{dyn} = 10/50 = 0.2$
b.	$\text{Power}_{st}/\text{Power}_{dyn} = 60/90 = 0.66$

1.9.4 $\text{Power}_{st}/\text{Power}_{dyn} = 0.6 \Rightarrow \text{Power}_{st} = 0.6 \times \text{Power}_{dyn}$

a.	$\text{Power}_{st} = 0.6 \times 35\text{ W} = 21\text{ W}$
b.	$\text{Power}_{st} = 0.6 \times 30\text{ W} = 18\text{ W}$

1.9.5

	1.2 V	1.0 V	0.8 V
a.	$P_{st} = 12.5$ W $P_{dyn} = 62.5$ W	$P_{st} = 10$ W $P_{dyn} = 50$ W	$P_{st} = 5.8$ W $P_{dyn} = 29.2$ W
b.	$P_{st} = 24.8$ W $P_{dyn} = 37.2$ W	$P_{st} = 20$ W $P_{dyn} = 30$ W	$P_{st} = 12$ W $P_{dyn} = 18$ W

1.9.6

a.	29.15
b.	23.32

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	2048	4096
	4	1024	4096
	8	512	4096

1.10.2

a.	Processors	Execution Time (μs)
	1	4.096
	2	2.368
	4	1.504
	8	1.152

b.	Processors	Execution Time (μs)
	1	4.096
	2	2.688
	4	1.664
	8	0.992

1.10.3

a.	Processors	Execution Time (μs)
	1	5.376
	2	3.008
	4	1.824
	8	1.312

b.	Processors	Execution Time (μs)
	1	5.376
	2	3.328
	4	1.984
	8	1.152

1.10.4

a.	Cores	Execution Time (s) @ 3 GHz
	1	4.00
	2	2.33
	4	1.50
	8	1.08

b.	Cores	Execution Time (s) @ 3 GHz
	1	3.33
	2	2.00
	4	1.16
	8	0.71

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	CPI for 1 Core
	1	1.2
	2	0.7
	4	0.45
	8	0.32

AQ 4

b.	Processors	CPI for 1 Core
	1	1
	2	0.6
	4	0.35
	8	0.21

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 $= \pi \times 10^2 = 314.2$ cm^2

Die area = wafer area/dies per wafer

a.	Die area = 176.7/84 = 2.10 cm^2
b.	Die area = 314.2/100 = 3.14 cm^2

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

a.	Yield = 0.96
b.	Yield = 0.91

1.11.2 Cost per die = cost per wafer/(dies per wafer × yield)

a.	Cost per die = 0.15
b.	Cost per die = 0.16

1.11.3

a.	Dies per wafer = 1.1 × 84 = 92 Defects per area = 1.15 × 0.02 = 0.023 defects/cm^2 Die area = wafer area/Dies per wafer = 176.7/92 = 1.92 cm^2 Yield = 0.96
b.	Dies per wafer = 1.1 × 100 = 110 Defects per area = 1.15 × 0.031 = 0.036 defects/cm^2 Die area = wafer area/Dies per wafer = 314.2/110 = 2.86 cm^2 Yield = 0.90

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

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

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 × CPU time/instr count

clock rate = 1/cycle time = 3 GHz

a.	CPI(bzip2) = $3 \times 10^9 \times 750/(2{,}389 \times 10^9) = 0.94$
b.	CPI(go) = $3 \times 10^9 \times 700/(1{,}658 \times 10^9) = 1.26$

1.12.2 SPECratio = ref. time/execution time

a.	SPECratio(bzip2) = 9,650/750 = 12.86
b.	SPECratio(go) = 10,490/700 = 14.98

1.12.3

$(12.86 \times 14.98)^{1/2} = 13.88$

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

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

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

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

CPU times(after)/CPU time(before) = 1.1 × 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. Thus, the SPECratio is decreased by 14%.

Solution 1.13

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

a.	CPI = $700 \times 4 \times 10^9/(0.85 \times 2{,}389 \times 10^9) = 1.37$
b.	CPI = $620 \times 4 \times 10^9/(0.85 \times 1{,}658 \times 10^9) = 1.75$

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

a.	CPI @ 4 GHz = 1.37, CPI @ 3 GHz = 0.94, ratio = 1.45
b.	CPI @ 4 GHz = 1.75, CPI @ 3 GHz = 1.26, ratio = 1.38

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.	700/750 = 0.933. CPU time reduction: 6.7%
b.	620/700 = 0.886. CPU time reduction: 11.4%

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

a.	No. instr = $960 \times 0.9 \times 4 \times 10^9/1.61 = 2{,}146 \times 10^9$
b.	No. instr = $690 \times 0.9 \times 4 \times 10^9/1.79 = 1{,}387 \times 10^9$

1.13.5 Clock rate = no. instr × CPI/CPU time.

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

1.13.6 Clock rate = no. instr × CPI/CPU time.

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

Solution 1.14

1.14.1 No. instr = 10^6

a.	$T(P1) = 5 \times 10^6 \times 0.9/(4 \times 10^9) = 1.125 \times 10^{-3}$ s $T(P2) = 10^6 \times 0.75/(3 \times 10^9) = 0.25 \times 10^{-3}$ s clock rate (P1) > clock rate (P2), performance (P1) < performance (P2)
b.	$T(P1) = 3 \times 10^6 \times 1.1/(3 \times 10^9) = 1.1 \times 10^{-3}$ s $T(P2) = 0.5 \times 10^6 \times 1/(2.5 \times 10^9) = 0.2 \times 10^{-3}$ s clock rate (P1) > clock rate (P2), performance (P1) < performance (P2)

1.14.2

a.	10^6 instructions, T(P1) = No. Intr × CPI/clock rate $T(P1) = 2.25 \times 10^{-4}$ s $T(P2) = N \times 0.75/(3 \times 10^9)$ then $N = 9 \times 10^5$
b.	10^6 instructions, T(P1) = No. Intr × CPI/clock rate $T(P1) = 3.66 \times 10^{-4}$ s $T(P2) = N \times 1/(3 \times 10^9)$ then $N = 9.15 \times 10^5$

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

a.	$MIPS(P1) = 4 \times 10^9 \times 10^{-6}/0.9 = 4.44 \times 10^3$ $MIPS(P2) = 3 \times 10^9 \times 10^{-6}/0.75 = 4.0 \times 10^3$ MIPS(P1) > MIPS(P2), performance(P1) < performance(P2) (from 1.14.1)
b.	$MIPS(P1) = 3 \times 10^9 \times 10^{-6}/1.1 = 2.72 \times 10^3$ $MIPS(P2) = 2.5 \times 10^9 \times 10^{-6}/1 = 2.5 \times 10^3$ MIPS(P1) > MIPS(P2), performance(P1) < performance(P2) (from 1.14.1)

1.14.4 MFLOPS = No. FP operations × 10^{-6}/T

a.	$T(P1) = (5 \times 10^5 \times 0.75 + 4 \times 10^5 \times 1 + 10 \times 10^5 \times 1.5)/(4 \times 10^9) = 5.86 \times 10^{-4}$ s $MFLOPS(P1) = 4 \times 10^5 \times 10^{-6}/(5.86 \times 10^{-4}) = 6.82 \times 10^2$ $T(P2) = (2 \times 10^6 \times 1.25 + 2 \times 10^6 \times 0.8 + 1 \times 10^6 \times 1.25)/(3 \times 10^9) = 1.78 \times 10^{-3}$ s $MFLOPS(P1) = 3 \times 10^5 \times 10^{-6}/(1.78 \times 10^{-3}) = 1.68 \times 10^2$
b.	$T(P1) = (1.5 \times 10^6 \times 1.5 + 1.5 \times 10^6 \times 1 + 2 \times 10^6 \times 2)/(4 \times 10^9) = 1.93 \times 10^{-3}$ s $MFLOPS(P1) = 1.5 \times 10^6 \times 10^{-6}/(1.93 \times 10^{-3}) = 0.77 \times 10^2$ $T(P2) = (0.8 \times 10^6 \times 1.25 + 0.6 \times 10^6 \times 1 + 0.6 \times 10^6 \times 2.5)/(3 \times 10^9) = 1.03 \times 10^{-3}$ s $MFLOPS(P2) = 0.6 \times 10^6 \times 10^{-6}/(1.03 \times 10^{-3}) = 5.82 \times 10^2$

1.14.5

a.	$T(P1) = (5 \times 10^5 \times 0.75 + 4 \times 10^5 \times 1 + 10 \times 10^5 \times 1.5)/(4 \times 10^9) = 5.86 \times 10^{-4}$ s $CPI(P1) = 5.86 \times 10^{-4} \times 4 \times 10^9/10^6 = 2.27$ $MIPS(P1) = 4 \times 10^9/(2.27 \times 10^6) = 1.76 \times 10^3$ $T(P2) = (2 \times 10^6 \times 1.25 + 2 \times 10^6 \times 0.8 + 1 \times 10^6 \times 1.25)/(3 \times 10^9) = 1.78 \times 10^{-3}$ s $CPI(P2) = 1.78 \times 10^{-3} \times 3 \times 10^9/(5 \times 10^6) = 1.068$ s $MIPS(P2) = 3 \times 10^9/(1.068 \times 10^6) = 2.78 \times 10^3$
b.	$T(P1) = (1.5 \times 10^6 \times 1.5 + 1.5 \times 10^6 \times 1 + 2 \times 10^6 \times 2)/(4 \times 10^9) = 1.93 \times 10^{-3}$ s $CPI(P1) = 1.93 \times 10^{-3} \times 4 \times 10^9/(5 \times 10^6) = 1.54$ $MIPS(P1) = 4 \times 10^9/(1.54 \times 10^6) = 2.59 \times 10^3$ $T(P2) = (0.8 \times 10^6 \times 1.25 + 0.6 \times 10^6 \times 1 + 0.6 \times 10^6 \times 2.5)/(3 \times 10^9) = 1.03 \times 10^{-3}$ s $CPI(P2) = 1.03 \times 10^{-3} \times 3 \times 10^9/(2 \times 10^6) = 1.54$ $MIPS(P1) = 3 \times 10^9/(1.54 \times 10^6) = 1.94 \times 10^3$

1.14.6

a.	$T(P1) = 5.86 \times 10^{-4}$ s (see problem 1.14.5) $performance(P1) = 1/T(P1) = 1.7 \times 10^3$ $T(P2) = 1.78 \times 10^{-3}$ s (see problem 1.14.5) $performance(P2) = 1/T(P2) = 5.6 \times 10^2$ perf(P1) > perf(P2), MIPS(P1) > MIPS(P2), MFLOPS(P1) < MFLOPS(P2)
b.	$T(P1) = 1.93 \times 10^{-3}$ s (see problem 1.14.5) $performance(P1) = 1/T(P1) = 5.1 \times 10^2$ $T(P2) = 1.03 \times 10^{-3}$ s (see problem 1.14.5) $performance(P2) = 1/T(P2) = 9.7 \times 10^2$ perf(P1) < perf(P2), MIPS(P1) < MIPS(P2), MFLOPS(P1) > MFLOPS(P2)

Solution 1.15

1.15.1

a.	$T_{fp} = 70 \times 0.8 = 56$ s. $T_{new} = 56 + 85 + 55 + 40 = 236$ s. Reduction: 5.6%
b.	$T_{fp} = 40 \times 0.8 = 32$ s. $T_{new} = 32 + 90 + 60 + 20 = 202$ s. Reduction: 3.8%

1.15.2

a.	$T_{new} = 250 \times 0.8 = 200$ s, $T_{fp} + T_{l/s} + T_{branch} = 165$ s, $T_{int} = 35$ s. Reduction time INT: 58.8%
b.	$T_{new} = 210 \times 0.8 = 168$ s, $T_{fp} + T_{l/s} + T_{branch} = 120$ s, $T_{int} = 48$ s. Reduction time INT: 46.6%

1.15.3

a.	$T_{new} = 250 \times 0.8 = 200$ s, $T_{fp} + T_{int} + T_{l/s} = 210$ s. NO
b.	$T_{new} = 210 \times 0.8 = 168$ s, $T_{fp} + T_{int} + T_{l/s} = 190$ s. NO

1.15.4

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

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

a.	2 processors: clock cycles = $4{,}096 \times 10^6$; $T_{cpu} = 2.048$ s
b.	16 processors: clock cycles = 512×10^6; $T_{cpu} = 0.256$ s

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

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

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

a.	2 processors: $CPI_{improved\ fp} = (2{,}048 - 3{,}816)/280 < 0$ ==> not possible
b.	16 processors: $CPI_{improved\ fp} = (256 - 462)/50 < 0$ ==> 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:

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

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

a.	2 processors: $CPI_{improved\ l/s} = (2{,}048 - 1{,}536)/640 = 0.8$
b.	16 processors: $CPI_{improved\ l/s} = (256 - 198)/80 = 0.725$

1.15.6

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

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

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

a.	2 processors: T_{cpu} (before improv.) = 2.048 s; T_{cpu} (after improv.) = 1.370 s
b.	16 processors: T_{cpu} (before improv.) = 0.256 s; T_{cpu} (after improv.) = 0.171 s

Solution 1.16

1.16.1 Without reduction in any routine:

a.	total time 4 proc = 102 ms
b.	total time 32 proc = 18 ms

Reducing time in routines A, C, and E:

a.	4 proc: T(A) = 10.2 ms, T(C) = 5.1 ms, T(E) = 2.5 ms, total time = 98.8 ms ==> reduction = 3.1%
b.	32 proc: T(A) = 1.7 ns, T(C) = 0.85 ns, T(E) = 1.7 ms, total time = 17.2 ms ==> reduction = 4.4%

1.16.2

a.	4 proc: T(B) = 40.5 ms, total time = 97.5 ms ==> reduction = 4.4%
b.	32 proc: T(B) = 6.3 ms, total time = 17.3 ms ==> reduction = 3.8%

1.16.3

a.	4 proc: T(D) = 32.4 ms, total time = 98.4 ms ==> reduction = 3.5%
b.	32 proc: T(D) = 5.4 ms, total time = 17.4 ms ==> reduction = 3.3%

1.16.4

AQ 5

No. Processors	Computing Time	Computing Time Ratio	Routing Time Ratio
2	201 ms		
4	131 ms	0.65	1.18
8	85 ms	0.65	1.31
16	56 ms	0.66	1.29
32	35 ms	0.62	1.05
64	18.5 ms	0.53	1.13

1.16.5 Geometric mean of computing time ratios = 0.62. Multiplying this by the computing time for a 64-processor system gives a computing time for a 128-processor system of 11.474 ms.

Geometric mean of routing time ratios = 1.19. Multiplying 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 = 201/0.62 = 324 ms. Routing time = 0, since no communication is required.

Author Query

AQ 1: Page S2: As meant t/o?
AQ 2: Page S3: As meant t/o?
AQ 3: Page S4: Close up t/o?
AQ 4: Page S12: Inserted heading OK?
AQ 5: Page S18: Blank cells as meant?

2 Solutions

Solution 2.1

2.1.1

a.	`sub f, g, h`
b.	`addi f, h, −5 (note, no subi)` `add f, f, g`

2.1.2

a.	1
b.	2

2.1.3

a.	−1
b.	0

2.1.4

a.	`f = f + 4`
b.	`f = g + h + i`

2.1.5

a.	5
b.	9

Solution 2.2

2.2.1

a.	`sub f, g, f`
b.	`addi f, h, −2 (note no subi)` `add f, f, i`

2.2.2

a.	1
b.	2

2.2.3

a.	1
b.	2

2.2.4

a.	`f += 4;`
b.	`f = i – (g + h);`

2.2.5

a.	5
b.	–1

Solution 2.3

2.3.1

a.	`sub f, $0, f` `sub f, f, g`

b.	`sub f, $0, f`
	`addi f, f, –5 (note, no subi)`
	`add f, f, g`

2.3.2

a.	2
b.	3

2.3.3

a.	–3
b.	–3

2.3.4

a.	`f += –4`
b.	`f += (g + h);`

2.3.5

a.	–3
b.	6

Solution 2.4

2.4.1

a.	`lw   $s0, 16($s6)`
	`sub  $s0, $0,  $s0`
	`sub  $s0, $s0, $s1`

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

2.4.2

a.	3
b.	4

2.4.3

a.	3
b.	6

2.4.4

a.	`f = 2j + i + g;`
b.	`B[g] = A[f] + A[1+f];`

2.4.5

a.	`slli $s2, $s4, 1`
	`add  $s0, $s2, $s3`
	`add  $s0, $s0, $s1`

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

2.4.6

a.	5 as written, 5 minimally
b.	7 as written, 6 minimally

Solution 2.5

2.5.1

a.	Address Data
	20 4
	24 5
	28 3
	32 2
	34 1
		`temp = Array[0];`
		`temp2 = Array[1];`
		`Array[0] = Array[4];`
		`Array[1] = Array[3];`
		`Array[3] = temp;`
		`Array[4] = temp2;`

b.	Address Data
	24 2
	38 4
	32 3
	36 6
	40 1
		`temp = Array[0];`
		`temp2 = Array[1];`
		`Array[0] = Array[4];`
		`Array[1] = temp;`
		`Array[4] = Array[3];`
		`Array[3] = temp2;`

2.5.2

a.	Address Data
	20 4
	24 5
	28 3
	32 2
	34 1
		`temp = Array[0];`
		`temp2 = Array[1];`
		`Array[0] = Array[4];`
		`Array[1] = Array[3];`
		`Array[3] = temp;`
		`Array[4] = temp2;`
			`lw   $t0, 0($s6)`
			`lw   $t1, 4($s6)`
			`lw   $t2, 16($s6)`
			`sw   $t2, 0($s6)`
			`lw   $t2, 12($s6)`
			`sw   $t2, 4($s6)`
			`sw   $t0, 12($s6)`
			`sw   $t1, 16($s6)`

b.	Address Data
	24 2
	38 4
	32 3
	36 6
	40 1
		`temp = Array[0];`
		`temp2 = Array[1];`
		`Array[0] = Array[4];`
		`Array[1] = temp;`
		`Array[4] = Array[3];`
		`Array[3] = temp2;`
			`lw $t0, 0($s6)`
			`lw $t1, 4($s6)`
			`lw $t2, 16($s6)`
			`sw $t2, 0($s6)`
			`sw $t0, 4($s6)`
			`lw $t0, 12($s6)`
			`sw $t0, 16($s6)`
			`sw $t1, 12($s6)`

2.5.3

a.	Address Data
	20 4
	24 5
	28 3
	32 2
	34 1
		`temp = Array[1];`
		`Array[1] = Array[5];`
		`Array[5] = temp;`
		`temp = Array[2];`
		`Array[2] = Array[4];`
		`temp2 = Array[3];`
		`Array[3] = temp;`
		`Array[4] = temp2;`
			`lw $t0, 0($s6)`
			`lw $t1, 4($s6)`
			`lw $t2, 16($s6)`
			`sw $t2, 0($s6)`
			`lw $t2, 12($s6)`
			`sw $t2, 4($s6)`
			`sw $t0, 12($s6)`
			`sw $t1, 16($s6)`
				8 MIPS instructions, +1 MIPS inst. for every non-zero offset lw/sw pair (11 MIPS inst.)

b.	Address Data
	24 2
	38 4
	32 3
	36 6
	40 1
		`temp = Array[3];`
		`Array[3] = Array[2];`
		`Array[2] = Array[1];`
		`Array[1] = Array[0];`
		`Array[0] = temp;`
			`lw $t0, 0($s6)`
			`lw $t1, 4($s6)`
			`lw $t2, 16($s6)`
			`sw $t2, 0($s6)`
			`sw $t0, 4($s6)`
			`lw $t0, 12($s6)`
			`sw $t0, 16($s6)`
			`sw $t1, 12($s6)`
				8 MIPS instructions, +1 MIPS inst. for every non-zero offset lw/sw pair (11 MIPS inst.)

2.5.4

a.	2882400018
b.	270544960

2.5.5

	Little-Endian	Big-Endian
a.	Address Data 12 ab 8 cd 4 ef 0 12	Address Data 12 12 8 ef 4 cf 0 ab
b.	Address Data 12 10 8 20 4 30 0 40	Address Data 12 40 8 30 4 20 0 10

Solution 2.6

2.6.1

a.	`lw $t0, 4($s7) # $t0 <-- B[1]`
	`sub $t0, $t0, $s1 # $t0 <-- B[1] – g`
	`add $s0, $t0, $s2 # f <-- B[1] –g + h`

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

2.6.2

a.	3
b.	6

2.6.3

a.	5
b.	4

2.6.4

a.	`f = f - i;`
b.	`f = 2 * (&A);`

2.6.5

a.	$s0 = –30
b.	$s0 = 512

2.6.6

a.

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

b.

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

Solution 2.7

2.7.1

a.	613566756
b.	1606303744

2.7.2

a.	613566756
b.	1606303744

2.7.3

a.	24924924
b.	5FBE4000

2.7.4

a.	11111111111111111111111111111111
b.	10000000000

2.7.5

a.	FFFFFFFF
b.	400

2.7.6

a.	1
b.	FFFFFC00

Solution 2.8

2.8.1

a.	`50000000, overflow`
b.	`0, no overflow`

2.8.2

a.	`B0000000, no overflow`
b.	`2, no overflow`

2.8.3

a.	`D0000000, overflow`
b.	`000000001, no overflow`

2.8.4

a.	overflow
b.	overflow

2.8.5

a.	overflow
b.	overflow

2.8.6

a.	overflow
b.	overflow

Solution 2.9

2.9.1

a.	no overflow
b.	overflow

2.9.2

a.	no overflow
b.	no overflow

2.9.3

a.	no overflow
b.	no overflow

2.9.4

a.	overflow
b.	overflow

2.9.5

a.	`94924924`
b.	`CFBE4000`

2.9.6

a.	2492614948
b.	–809615360

Solution 2.10

2.10.1

a.	`add  $s0, $s0, $s0`
b.	`sub  $t1, $t2, $t3`

2.10.2

a.	r-type
b.	r-type

2.10.3

a.	`2108020`
b.	`14B4822`

2.10.4

a.	`0x21080001`
b.	`0xAD490020`

2.10.5

a.	i-type
b.	i-type

2.10.6

a.	`op=0x8, rs=0x8, rs=0x8, imm=0x0`
b.	`op=0x2B, rs=0xA, rt=0x9, imm=0x20`

Solution 2.11

2.11.1

a.	$0000\ 0001\ 0000\ 1000\ 0100\ 0000\ 0010\ 0000_{two}$
b.	$0000\ 0010\ 0101\ 0011\ 1000\ 1000\ 0010\ 0010_{two}$

2.11.2

a.	17317920
b.	39028770

2.11.3

a.	`add $t0, $t0, $t0`
b.	`sub $s1, $s2, $s3`

2.11.4

a.	r-type
b.	i-type

2.11.5

a.	`sub $v1, $v1, $v0`
b.	`lw $v0, 4($at)`

2.11.6

a.	`0x00621822`
b.	`0x8C220004`

Solution 2.12

2.12.1

	Type	opcode	rs	rt	rd	shamt	funct	
a.	r-type	6	7	7	7	5	6	total bits = 38
b.	r-type	8	5	5	5	5	6	total bits = 34

2.12.2

	Type	opcode	rs	rt	immed	
a.	i-type	6	7	7	16	total bits = 36
b.	i-type	8	5	5	16	total bits = 34

2.12.3

a.	more registers → more bits per instruction → could increase code size more registers → less register spills → less instructions
b.	more instructions → more appropriate instruction → decrease code size more instructions → larger opcodes → larger code size

2.12.4

a.	17367058
b.	2903048210

2.12.5

a.	`sub $t0, $t1, $0`
b.	`sw $t1, 12($t0)`

2.12.6

a.	r-type, op=0x0, rt=0x9
b.	i-type, op=0x2B, rt=0x8

Solution 2.13

2.13.1

a.	0xBABEFEF8
b.	0x11D111D1

2.13.2

a.	0xAAAAAAA0
b.	0x00DD00D0

2.13.3

a.	0x00005545
b.	0x0000BA01

2.13.4

a.	0x00014B4A
b.	0x00000001

2.13.5

a.	0x4b4a0000
b.	0x00000000

2.13.6

a.	0x4b4bfffe
b.	0x0000003C

Solution 2.14

2.14.1

a.	lui $t1, 0x003f ori $t1, $t0, 0xffe0 and $t1, $t0, $t1 srl $t1, $t1, 5
b.	lui $t1, 0x003f ori $t1, $t0, 0xffe0 and $t1, $t0, $t1 sll $t1, $t1, 9

2.14.2

a.	add $t1, $t0, $0 sll $t1, $t1, 28
b.	andi $t0, $t0, 0x000f sll $t0, $t0, 14 ori $t1, $t1, 0x3fff sll $t1, $t1, 18 ori $t1, $t1, 0x3fff or $t1, $t1, $t0

2.14.3

a.	srl $t1, $t0, 28 sll $t1, $t1, 29
b.	srl $t0, $t0, 28 andi $t0, $t0, 0x0007 sll $t0, $t0, 14 ori $t1, $t1, 0x7fff sll $t1, $t1, 17 ori $t1, $t1, 0x3fff or $t1, $t1, $t0

2.14.4

a.	srl $t0, $t0, 11 sll $t0, $t0, 26 ori $t2, $0, 0x03ff sll $t2, $t2, 16 ori $t2, $t2, 0xffff and $t1, $t1, $t2 or $t1, $t1, $t0
b.	srl $t0, $t0, 11 sll $t0, $t0, 26 srl $t0, $t0, 12 ori $t2, $0, 0xfff0 sll $t2, $t2, 16 ori $t2, $t2, 0x3fff and $t1, $t1, $t2 or $t1, $t1, $t0

2.14.5

a.	`sll $t0, $t0, 27`
	`ori $t2, $0, 0x07ff`
	`sll $t2, $t2, 16`
	`ori $t2, $t2, 0xffff`
	`and $t1, $t1, $t2`
	`or $t1, $t1, $t0`

b.	`sll $t0, $t0, 27`
	`srl $t0, $t0, 13`
	`ori $t2, $0, 0xfff8`
	`sll $t2, $t2, 16`
	`ori $t2, $t2, 0x3fff`
	`and $t1, $t1, $t2`
	`or $t1, $t1, $t0`

2.14.6

a.	`srl $t0, $t0, 29`
	`sll $t0, $t0, 30`
	`ori $t2, $0, 0x3fff`
	`sll $t2, $t2, 16`
	`ori $t2, $t2, 0xffff`
	`and $t1, $t1, $t2`
	`or $t1, $t1, $t0`

b.	`srl $t0, $t0, 29`
	`sll $t0, $t0, 30`
	`srl $t0, $t0, 16`
	`ori $t2, $0, 0xffff`
	`sll $t2, $t2, 16`
	`ori $t2, $t2, 0x3fff`
	`and $t1, $t1, $t2`
	`or $t1, $t1, $t0`

Solution 2.15

2.15.1

a.	0xff005a5a
b.	0x00ffffe7

2.15.2

a.	`nor $t1, $t2, $t2`
b.	`nor $t1, $t3, $t3` `or $t1, $t2, $t1`

2.15.3

a.	`nor $t1, $t2, $t2`	000000 01010 01010 01001 00000 100111
b.	`nor $t1, $t3, $t3` `or $t1, $t2, $t1`	000000 01011 01011 01001 00000 100111 000000 01010 01001 01001 00000 100101

2.15.4

a.	0xFFFFFFFF
b.	0x00012340

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

a.	`nor $t3, $t1, $t1` `or $t1, $t2, $t3`
b.	`lw $t3, 0($s1)` `sll $t1, $t3, 4`

2.15.6

a.	`nor $t3, $t1, $t1` `or $t1, $t2, $t3`	000000 01001 01001 01011 00000 100111 000000 01010 01011 01001 00000 100101
b.	`lw $t3, 0($s1)` `sll $t1, $t3, 4`	100011 10001 01011 0000000000000000 000000 00000 01011 01001 00100 000000

Solution 2.16

2.16.1

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

2.16.2

a.	none
b.	none

2.16.3

a.	Jump – No, Beq - No
b.	Jump – No, Beq - No

2.16.4

a.	$t2 = 2
b.	$t2 = 1

2.16.5

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

2.16.6

a.	jump – Yes, beq - no
b.	jump – no, beq - no

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.	i-type
b.	i-type

2.17.3

a.	`addi $t2, $t3, −5`
b.	`addi $t2, $t2, −1` `beq $t2, $0, loop`

2.17.4

a.	20
b.	20

2.17.5

a.

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

b. Same as part a.

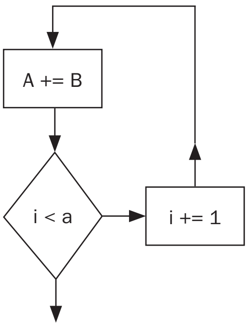
2.17.6

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

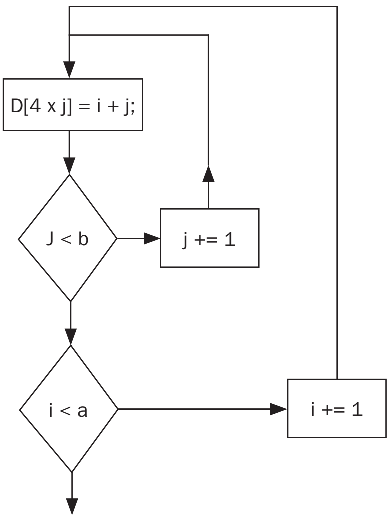
Solution 2.18

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: slt  $t2, $t0, $s0
      bne  $t2, $0, LOOP
```

b.	<pre> addi $t0, $0, 0
	      beq $0, $0, TEST1
	LOOP1:addi $t1, $0, 0
	      beq $0, $0, TEST2
	LOOP2:add $t3, $t0, $t1
	      sll $t2, $t1, 4
	      add $t2, $t2, $s2
	      sw $t3, ($t2)
	      addi $t1, $t1, 1
	TEST2:slt $t2, $t1, $s1
	      bne $t2, $0, LOOP2
	      addi $t0, $t0, 1
	TEST1:slt $t2, $t0, $s0
	      bne $t2, $0, LOOP1</pre>

2.18.3

a.	6 instructions to implement and infinite instructions executed
b.	14 instructions to implement and 158 instructions executed

2.18.4

a.	351
b.	601

2.18.5

a.	<pre>for(i=50; i>0; i--){
	    result += MemArray[s0];
	    result += MemArray[s0+1];
	    s0 += 2;
	}</pre>

b.	<pre>for (i=0; i<100; i++) {
	    result += MemArray[s0];
	    s0 = s0 + 4;
	}</pre>

2.18.6

a.	<pre> addi $t1, $s0, 400
	LOOP: lw $s1, 0($s0)
	      add $s2, $s2, $s1
	      lw $s1, 4($s0)
	      add $s2, $s2, $s1
	      addi $s0, $s0, 8
	      bne $s0, $t1, LOOP</pre>

b.	<pre> addi $t1, $s0, 400
	LOOP: lw $s1, 0($t1)
	      add $s2, $s2, $s1
	      addi $t1, $t1, -4
	      bne $t1, $s0, LOOP</pre>

Solution 2.19

2.19.1

a.

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

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

b.

```
positive:
      addi $sp, $sp, -4
      sw   $ra, 0($sp)
      jal  addit
      addi $t1, $0, 1
      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
addit:
      add  $v0, $a0, $a1
      jr   $ra
# 13  instructions worst-case
```

2.19.2

a.	Due to the recursive nature of the code, not possible for the compiler to in-line the function call.

b.	<pre>positive:
	      add $t0, $a0, $a1
	      addi $v0, $0, 1
	      slt $t2, $0, $t0
	      bne $t2, $0, exit
	      addi $v0, $0, $0
	exit:
	      jr $ra
	
	# 6 instructions worst-case</pre>

2.19.3

a.	<pre>after calling function fib:
	old $sp -> 0x7ffffffc ???
	                 -4 contents of register $ra for fib(N)
	                 -8 contents of register $s0 for fib(N)
	$sp-> -12 contents of register $a0 for fib(N)
	
	there will be N-1 copies of $ra, $s0, and $a0</pre>

b.	<pre>after calling function positive:
	old $sp -> 0x7ffffffc ???
	$sp-> -4 contents of register $ra
	
	after calling function addit:
	old $sp -> 0x7ffffffc ???
	                -4 contents of register $ra
	$sp-> -8 contents of register $ra #return to
	positive</pre>

2.19.4

a.	<pre>f: addi $sp,$sp,-12
	    sw $ra,8($sp)
	    sw $s1,4($sp)
	    sw $s0,0($sp)
	    move $s1,$a2
	    move $s0,$a3
	    jal func
	    move $a0,$v0
	    add $a1,$s0,$s1
	    jal func
	    lw $ra,8($sp)
	    lw $s1,4($sp)
	    lw $s0,0($sp)
	    addi $sp,$sp,12
	    jr $ra</pre>

b.	```
	f: addi $sp,$sp,–4
	    sw $ra,0($sp)
	    add $t0,$a1,$a0
	    add $a1,$a3,$a2
	    slt $t1,$a1,$t0
	    beqz $t1,L
	    move $a0,$t0
	    jal func
	    lw $ra,0($sp)
	    addi $sp,$sp,4
	    jr ra
	L: move $a0,$a1
	    move $a1,$t0
	    jal func
	    lw $ra,0($sp)
	    addi $sp,$sp,4
	    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, $s0, $s1, and $sp before that call. We save only one instruction (jr $ra).
b.	We can use the tail-call optimization for either call to `func` (when the condition for the if is true or false). This eliminates the need to save $ra and move the stack pointer, so we execute 5 fewer instructions (regardless of whether the if condition is true or not). The code of the function is 8 instructions shorter because we can eliminate both instances of the code that restores $ra and returns.

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 function `f` was called, and register $t5 can have an arbitrary value. For register $t5, note that although our function `f` does not modify it, function `func` is allowed to modify it so we cannot assume anything about the value of $t5 after function func has been called.

Solution 2.20

2.20.1

a.

```
FACT:   addi $sp, $sp, –8  # make room in stack for 2 more items
        sw   $ra, 4($sp)   # save the return address
        sw   $a0, 0($sp)   # save the argument n
        slti $t0, $a0, 1   # $t0 = $a0 x 2
        beq, $t0, $0, L1   # if $t0 = 0, goto L1
        add  $v0, $0, 1    # return 1
        add  $sp, $sp, 8   # pop two items from the stack
        jr   $ra           # return to the instruction after jal
L1:     addi $a0, $a0, –1  # subtract 1 from argument
        jal  FACT          # call fact(n–1)
        lw   $a0, 0($sp)   # just returned from jal: restore n
        lw   $ra, 4($sp)   # restore the return address
        add  $sp, $sp, 8   # pop two items from the stack
        mul  $v0, $a0, $v0 # return n*fact(n–1)
        jr   $ra           # return to the caller
```

b.	`FACT: addi $sp, $sp, -8     # make room in stack for 2 more items`
	`      sw   $ra, 4($sp)      # save the return address`
	`      sw   $a0, 0($sp)      # save the argument n`
	`      slti $t0, $a0, 1      # $t0 = $a0 x 2`
	`      beq, $t0, $0, L1      # if $t0 = 0, goto L1`
	`      add  $v0, $0, 1       # return 1`
	`      add  $sp, $sp, 8      # pop two items from the stack`
	`      jr   $ra              # return to the instruction after jal`
	`L1:   addi $a0, $a0, -1     # subtract 1 from argument`
	`      jal  FACT             # call fact(n-1)`
	`      lw   $a0, 0($sp)      # just returned from jal: restore n`
	`      lw   $ra, 4($sp)      # restore the return address`
	`      add  $sp, $sp, 8      # pop two items from the stack`
	`      mul  $v0, $a0, $v0    # return n*fact(n-1)`
	`      jr   $ra              # return to the caller`

2.20.2

a.	25 MIPS instructions to execute non-recursive vs. 45 instructions to execute (corrected version of) recursion
	
	Non-recursive 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 non-recursive vs. 45 instructions to execute (corrected version of) recursion
	
	Non-recursive 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.	<pre>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</pre>

b.	<pre>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</pre>

2.20.5

<table>
<tr><td>a.</td><td>23 MIPS instructions to execute non-recursive vs. 73 instructions to execute (corrected version of) recursion

Non-recursive version:
<pre>
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
</pre></td></tr>
<tr><td>b.</td><td>23 MIPS instructions to execute non-recursive vs. 73 instructions to execute (corrected version of) recursion

Non-recursive version:
<pre>
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
</pre></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.	<pre>MAIN: addi $sp, $sp, −4
	        sw $ra, ($sp)
	
	        addi $a0, $0, 10
	        addi $a1, $0, 20
	        jal FUNC
	        add $t2, $v0 $0
	
	        lw $ra, ($sp)
	        addi $sp, $sp, 4
	        jr $ra
	
	FUNC: lw $t1, ($s0) #assume $s0 has global variable base
	        sub $t0, $v0, $v1
	        addi $v0, $t0, $t1
	        jr $ra</pre>

b.	<pre>MAIN: addi $sp, $sp, −4
	        sw $ra, ($sp)
	
	        lw $t1, ($s0) #assume $s0 has global variable base
	        addi $a0, $t1, 1
	        jal LEAF
	        add $t2, $v0 $0
	
	        lw $ra, ($sp)
	        addi $sp, $sp, 4
	        jr $ra
	
	LEAF: addi $v0, $a0, 1
	        jr $ra</pre>

2.21.2

a.	<pre>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</pre>

b.	`after entering function main:`
	`old $sp ->      0x7ffffffc     ???`
	`$sp->           -4             contents of register $ra`
	
	`global pointers:`
	`0x10008000      100            my_global`
	
	`after entering function leaf_function:`
	`old $sp ->      0x7ffffffc     ???`
	`                -4             contents of register $ra`
	`$sp->           -8              contents of register $ra (return to main)`
	
	`global pointers:`
	`0x10008000      101            my_global`

2.21.3

a.

```
MAIN:   addi $sp, $sp, -4
        sw   $ra, ($sp)

        addi $a0, $0, 10
        addi $a1, $0, 20
        jal  FUNC
        add  $t2, $v0 $0

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

FUNC:   lw   $t1, ($s0)   #assume $s0 has global variable base
        sub  $t0, $v0, $v1
        addi $v0, $t0, $t1
        jr   $ra
```

b.

```
MAIN:   addi $sp, $sp, -4
        sw   $ra, ($sp)

        lw   $t1, ($s0)   #assume $s0 has global variable base
        addi $a0, $t1, 1
        jal  LEAF
        add  $t2, $v0 $0

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

LEAF:   addi  $v0, $a0, 1
        jr   $ra
```

2.21.4

a.	The return address of the function is in $ra, so the last instruction should be "jr $ra."
b.	The tail call to g must use jr, not jal. If jal is used, it overwrites the return address so function g returns back to f, not to the original caller of f as intended.

2.21.5

a.

```
int f(int a, int b, int c){
  if(c)
    return (a+b);
  return (a−b);
}
```

b.

```
int f(int a, int b, int c, int d){
  if(a>c+d)
    return b;
  return g(b);
}
```

2.21.6

a.	The function returns 101 (1000 is nonzero, so it returns 1+100).
b.	The function returns 500 (c+d is 1030, which is larger than 1, so the function returns g(b), which according to the problem statement is 500).

Solution 2.22

2.22.1

a.	68 65 6C 6C 6F 20 77 6F 72 6C 64
b.	48 49 50 51 52 53 54 55 56 57

2.22.2

a.	U+0038, U+0020, U+0062, U+0069, U+0074, U+0073
b.	U+0030, U+0031, U+0032, U+0033, U+0034, U+0035, U+0036, U+0037, U+0038, U+0039

2.22.3

a.	ADD
b.	MIPS

Solution 2.23

2.23.1

a.	<pre>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</pre>

b.	<pre>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</pre>

Solution 2.24

2.24.1

a.	0x00000012
b.	0x0012ffff

2.24.2

a.	0x00000080
b.	0x00800000

2.24.3

a.	0x00000011
b.	0x00115555

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.	yes

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

a.	no
b.	no

2.25.4 Range is 0x1FFFF004 + 0x1FFFC = 0x2001F000 to 0x1FFFF004 – 0x20000 = 1FFDF004

a.	yes
b.	no

2.25.5 Generally, all solutions are similar:

```
add  $t1, $0, $0          #clear $t1
addi $t2, $0, 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, $0, 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, $0, 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.	0x00000000

2.25.7

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

Solution 2.26

2.26.1 Branch range is 0x00020000 to 0xFFFE0004.

a.	one branch
b.	one branch

2.26.2

a.	one
b.	can't be done

2.26.3 Branch range is 0x00000200 to 0xFFFFFE04.

a.	256 branches
b.	one branch

2.26.4

a.	branch range is 16× smaller
b.	branch range is 4× smaller

2.26.5

a.	no change
b.	jump to addresses 0 to 2^{26} instead of 0 to 2^{28}, assuming the PC<0x08000000

2.26.6

a.	rs field now 7 bits
b.	no change

Solution 2.27

2.27.1

a.	MIPS lw/sw instructions: lw $t0, 8($t1)
b.	jump

2.27.2

a.	i-type
b.	j-type

2.27.3

a.	+ allows memory from (base +/– 2^{15}) addresses to be loaded without changing the base – max size of 64 kB memory array without having to use multiple base addresses
b.	+ large jump range – jump range not as large as jump-register – can only access 1/16th of the total addressable space

2.27.4

a.	`0x00400000        beq   $s0, $0, FAR`
	`...`
	`0x00403100  FAR:  addi  $s0, $s0, 1`
		`0x12000c3c`
		
		`0x22100001`

b.	`0x00000100        j     AWAY`
	`...`
	`0x04000010  AWAY: addi  $s0, $s0, 1`
		`0x09000004`
		
		`0x22100001`

2.27.5

a.	<pre> addi $t0, $0, 0x31
	      sll $t0, $t0, 8
	      beq $s0, $0, TEMP
	      ...
	TEMP: jr $t0</pre>

b.	<pre> addi $s0, $0, 0x4
	      sll $s0, $s0, 24
	      ori $s0, $s0, 0x10
	      jr $s0
	      ...
	      addi $s0, $s0, 1</pre>

2.27.6

a.	2
b.	3

Solution 2.28

2.28.1

a.	3 instructions

2.28.2

a.	The location specified by the LL instruction is different than the SC instruction; hence, the operation of the store conditional is undefined.

2.28.3

a.	<pre>try: MOV R3,R4
	      LL R2,0(R1)
	      ADDI R2, R2, 1
	      SC R3,0(R1)
	      BEQZ R3,try
	      MOV R4,R2</pre>

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)	1	1	2	99	99	40
ll $t1, 0($s1)		2	99	2	99	99	40
	sc $t0, 0($s1)	3	99	2	40	99	1
sc $t0, 0($s1)		4	99	0	40	99	1

b.

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)		1	99	2	99	30	40
	ll $t1, 0($s1)	2	99	2	99	99	40
	addi $t1,$t1,1	3	99	2	99	100	40
	sc $t0, 0($s1)	4	99	2	100	100	1
sc $t0, 0($s1)		5	99	0	100	100	1

Solution 2.29

2.29.1 The critical section can be implemented as:

```
comment:  Not sure what this is...

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

       operation

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

Where operation is implemented as:

a.	`lw $t0,0($a1)`
	`slt $t1,$t0,$a2`
	`bne $t1,$0,skip`
	`sw $a2,0($a1)`
	`skip:`

b.	`lw $t0,0($a1)`
	`blez $t0,skip`
	`sle $t1,$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)`
	`sle $t1,$t0,$a2`
	`bnez $t1,skip`
	`mov $t0,$a2`
	`sc $t0,0($a1)`
	`beqz $t0,try`
	`skip:`

b.	`try: ll $t0,0($a1)`
	`blez $t0,skip`
	`sle $t1,$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 if it uses a register whose value is no longer needed. We have:

	Lock-based	Direct LL/SC implementation
a.	6 + 3	3
b.	6 + 2	2

2.29.4

a.	It is possible for one or both processors to complete this code without ever reaching the SC instruction. If only one executes SC, it completes successfully. If both reach SC, they do so in the same cycle, but one SC completes first and then the other detects this and fails.
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 first 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 cannot be shared. Therefore, shared variable `shvar` must be kept in memory, loaded each time its 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 first 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 first LL to fail (we have an LL and a 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 $t0, $0, $0`
b.	`add $t0, $0, large` `beq $t1, $t0, LOOP`

2.30.2

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

Solution 2.31

2.31.1

a.

	Text Size	0x440
	Data Size	0x90
Text	Address	Instruction
	0x00400000	lbu $a0, 8000($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
	…	…
	0x00400140	sw $a0, 8040($gp)
	0x00400144	jmp 0x04002C0
	…	…
	0x004002C0	jal 0x0400000
	…	…
Data	0x10000000	(X)
	…	…
	0x10000040	(Y)

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

2.31.3 The limitation on the sizes of the displacement and address fields 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:
lw      $v0,0($a0)
lw      $v1,0($a1)
sw      $v1,0($a0)
sw      $v0,0($a1)
jr      $ra
```

b.

```
swap:
lw      $t0,0($a0)
lw      $t1,0($a1)
add     $t0,$t0,$t1
sub     $t1,$t0,$t1
sub     $t0,$t0,$t1
sw      $t0,0($a0)
sw      $t1,0($a1)
jr      $ra
```

2.32.2

a.	Pass the address of v[j] and of v[j+1] to swap. Because the address of v[j] 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 "mov $a0,$t2" and "addi $a1,$t2,4."
b.	Pass the address of v[j] and of v[j+1] to swap. Because the address of v[j] 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 "mov $a0,$t2" and "addi $a1,$t2,4."

2.32.3

a.

```
swap:
lb      $v0,0($a0) ; Byte-sized load
lb      $v1,0($a1)
sb      $v1,0($a0) ; Byte-sized store
sb      $v0,0($a1)
jr      $ra
```

b.

```
swap:
lb      $t0,0($a0)  ; Byte-sized load
lb      $t1,0($a1)
add     $t0,$t0,$t1
sub     $t1,$t0,$t1
sub     $t0,$t0,$t1
sb      $t0,0($a0)  ; Byte-sized store
sb      $t1,0($a1)
jr      $ra
```

2.32.4

a.	No change to saving/restoring code is needed because the same s-registers are used in the modified sort() code.
b.	No change. This modification affects array address computation and load/store instructions. We still need to use the same s-registers which need to be saved/restored.

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

a.	The number of instructions in sort() is unchanged. The swap() function is changed, but it is never executed when sorting an already-sorted array. As a result, we execute exactly the same number of instructions.
b.	The only change in the number of instructions is that sll instructions can be eliminated in both sort() and swap(). When sorting an already-sorted array, swap() is never executed, and the inner loop in sort() always exits during its first iteration, so we save one sll 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.	The number of instructions in sort() is unchanged. However, the swap() function now has only 5 instructions (instead of 7) so we now execute 90 instructions fewer.

b.	One fewer instruction is executed each time v[j] is needed to check the "v[j]>v[j+1]" condition for the inner loop. This happens a total of 45 times. Also, swap() now has one instruction less (no sll is needed), so there we also execute a total of 45 fewer instructions. Overall, we execute 90 instructions fewer.

Solution 2.33

2.33.1

a.

```
copy: move $t0,$0
loop: beq  $t0,$a2,done
      sll  $t1,$t0,2
      add  $t2,$t1,$a1
      lw   $t2,0($t2)
      add  $t1,$t1,$a0
      sw   $t2,0($t1)
      addi $t0,$t0,1
      b    loop
done: jr   $ra
```

b.

```
shift:  move $t0,$0
        addi $t1,$a1,-1
loop:   beq  $t0,$t1,done
        sll  $t2,$t0,2
        add  $t2,$t2,$a0
        lw   $t3,4($t2)
        sw   $t3,0($t2)
        addi $t0,$t0,1
        b    loop
done:   jr $ra
```

2.33.2

a.

```
void copy(int *a, int *b, int n){
  int *p,*q;
  for(p=a,q=b;p!=a+n;p++,q++)
    *p=*q;
}
```

b.

```
void shift(int *a, int n){
  int *p;
  for(p=a;p!=a+n-1;p++)
    *p=*(p+1);
}
```

2.33.3

a.	`copy: move $t0,$a0`
	`      move $t1,$a1`
	`      sll  $t2,$a2,2`
	`      add  $t2,$t2,$a0`
	`loop: beq  $t0,$t2,done`
	`      lw   $t3,0($t1)`
	`      sw   $t3,0($t0)`
	`      addi $t0,$t0,4`
	`      addi $t1,$t1,4`
	`      b    loop`
	`done: jr   $ra`

b.	`find: move $t0,$a0`
	`      sll  $t1,$a1,2`
	`      add  $t1,$t1,$a0`
	`loop: beq  $t0,$t1,done`
	`      lw   $t2,4($t0)`
	`      sw   $t2,0($t0)`
	`skip: addi $t0,$t0,4`
	`      b    loop`
	`done: jr   $ra`

2.33.4

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

2.33.5

	Array-based	Pointer-based
a.	3	4
b.	4	3

2.33.6 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.	`add  $s0, $s1, $s2` `# no equivalent to ADC in MIPS`

b.	`addi $t0, $0, 4`
	`beq  $s0, $t0, LABEL`
	`add  $s1, $s1, $s0`

2.34.2

a.	ADD, ADC — both ARM register-register instruction format
b.	CMP, ADDNE — both ARM register-register instruction format

2.34.3

a.	`ORR   r0, 0`
	`NOT   r4, r0`
	`AND   r1, r4`

b.	`ROR   r1, r2, #16`

2.34.4

a.	ORR, NOT, AND — all ARM register-register instruction format
b.	ROR — an ARM register-register instruction format

Solution 2.35

2.35.1

a.	register + offset (displacement or based)
b.	rregister + offset and update register

2.35.2

a.	`addi $s1, $s1, 4` `lw   $s0, 4($s1)`

b.	`lw   $s1, 0($s0)`
	`lw   $s2, 4($s0)`
	`lw   $s3, 8($s0)`
	`addi $s0, $s0, 12`

2.35.3

a.	`        addi $s0, $0, 10`
	`LOOP:   add  $s0, $s0, $s1`
	`        addi $s0, $s0, -1`
	`        bne  $s0, $0, LOOP`

b.	`        addu $s0, $s0, $s1   # add lower words`
	`        sltu $t0, $s0, $s1   # find sign bit`
	`        addu $t0, $t0, $s2   # add sign bit to upper word`
	`        addu $s2, $t0, $s3   # add upper words`

2.35.4

a.	4 ARM vs. 4 MIPS instructions
b.	2 ARM vs. 4 MIPS instructions

2.35.5

a.	ARM 0.67 times as fast as MIPS
b.	ARM 1.33 times as fast as MIPS

Solution 2.36

2.36.1

a.	`srl  $s1, $s1, 4` `add  $s3, $s2, $s1`
b.	`add  $s3, $s2, $s1`

2.36.2

a.	`add  $s3, $s2, $0`
b.	`addi $s3, $s2, 8`

2.36.3

a.	`srl  $s1, $s1, 4` `add  $s3, $s2, $s1`
b.	`add  $s3, $s2, $s1`

2.36.4

a.	`ADD  r3, r2, #2`
b.	`SUBS r3, r2, -1`

Solution 2.37

2.37.1

a.	START: mov eax, 3
	push eax
	mov eax, 4
	mov ecx, 4
	add eax, ecx
	pop ecx
	add eax, ecx
		eax = (4 + 4) + 3

b.	START: mov ecx, 100
	mov eax, 0
	LOOP: add eax, ecx
	dec ecx
	cmp ecx, 0
	jne LOOP
	DONE:
		ebx = 0;
		for (i=100; i>0; i--)
		ebx += i

2.37.2

a.	START: addi $s0, $0, 3
	addi $sp, $sp, -4
	sw $s0, 0($sp)
	addi $s0, $0, 4
	addi $s2, $0, 4
	add $s0, $s0, $s2
	lw $s2, 0($sp)
	addi $sp, $sp, 4
	add $s0, $s0, $s2

b.	START: add $s0, $0, $0
	addi $s2, $0, 100
	LOOP: add $s0, $s0, $s2
	addi $s2, $s2, -1
	bne $s2, $0, LOOP

2.37.3

a.	push eax	5,3
b.	test eax, 0x00200010	7, 1, 8, 32

2.37.4

a.	sw $a0, 0($sp)
b.	addi $t0, $0, 0x00200010 and $t1, $s0, $t0 slt $t2, $t1, $0

Solution 2.38

2.38.1

a.	This instruction copies ECX elements, where each element is 2 bytes in size, from an array pointed to by ESI to an array pointer by EDI.
b.	This instruction finds the first occurrence of a byte (given in AL) in an array pointed to by EDI. The search stops when the byte is found, or when the entire length of the array (specified in ECX) is searched. For example, the C library function strlen can easily be implemented using this instruction.

2.38.2

a.

```
loop: lh   $t0,0($a2)
      sh   $t0,0($a1)
      addi $a0,$a0,-1
      addi $a1,$a1,2
      addi $a2,$a2,2
      bnez $a0,loop
```

b.

```
loop: lb   $t0,0($a1)
      beq  $t0,$a3,done
      addi $a0,$a0,-1
      addi $a1,$a1,1
      bnez $a0,loop
done:
```

2.38.3

	x86	MIPS	Speedup
a.	5	6	1.2
b.	3	5	1.67

2.38.4

a.

MIPS Code:

```
f: slt   $t0,$a1,$a0
   beqz  $t0,S
   move  $v0,$a2
   jr    $ra
S: move  $v0,$a3
   jr    $ra
```

Code Size Comparison:

MIPS: $6 \times 4 = 24$ bytes
x86: 25 bytes

b.

MIPS Code:

```
f: beqz  $a1,D
   move  $t0,zero
   move  $t1,$a0
L: addi  $t0,$t0,1
   sw    $0,0($t1)
   addi  $t1,$t1,4
   bne   $t0,$a1,L
D: jr    $ra
```

Code Size Comparison:

MIPS: $8 \times 4 = 32$ bytes
x86: 31 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 first one, so it is more difficult to design a processor that executes multiple instructions in parallel.

2.38.6 Under these assumptions, using x86 leads to a significant slowdown (the speedup is well below 1):

	MIPS Cycles	x86 Cycles	Speedup
a.	4	15	0.27
b.	2	13	0.15

Solution 2.39

2.39.1

a.	0.76 seconds
b.	2.86 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)

$$\begin{aligned}\text{new CPU time} = \; & 0.75 \times \text{old ICa} \times \text{CPIa} \times 1.1 \times \text{oldCCT} \\ & + \text{oldICls} \times \text{CPIls} \times 1.1 \times \text{oldCCT} \\ & + \text{oldICb} \times \text{CPIb} \times 1.1 \times \text{oldCCT}\end{aligned}$$

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

2.39.3

a.	107.04%	113.43%
b.	107.52%	114.4%

2.39.4

a.	2.6
b.	3.7

2.39.5

a.	0.88
b.	0.26

2.39.6

a.	0.533333333
b.	not possible

Solution 2.40

2.40.1

a.	In the first iteration $t0 points to a[0] and the lw fetches a[0] as intended. In the second iteration $t0 points to the next byte and the lw uses a non-aligned address and causes a bus error. Note that the computation for $t1 (address of a[n]) does not cause a bus error because that address is not actually used to access memory.
b.	In the very first iteration $0 is 0, and the address of the first lw is one byte into a[0] instead of a[1]. This means this access is non-aligned and causes a bus error.

2.40.2

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

2.40.3

a.

```
f: move $v0,$0
   move $t0,$a0
   sll  $t1,$a1,2      ; We must multiply n by 4 to get the address
   add  $t1,$t1,$a0    ; of the end of array a
L: lw   $t2,0($t0)
   bne  $t2,$a2,S
   addi $v0,$v0,1
S: addi $t0,$t0,4      ; Move to next element in a
   bne  $t0,$t1,L
   jr   $ra
```

b.	```f: move $t0,$0```
	``` addi $t1,$a1,-1```
	```L: sll $t2,$t0,2 ; We must multiply the index by 4 before we```
	``` add $t2,$t2,$a0 ; add it to a[] to form the address for lw```
	``` lw $t3,4($t2) ; The offset of a[i+1] from a[i] is 4, not 1```
	``` sw $t3,0($t2)```
	``` addi $t0,$t0,1```
	``` bne $t0,$t1,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.	10, 11, 12, 13, 14
b.	100, 102, 104, 106, 108

2.40.5 In main, register $s0 becomes 5, then my_alloc is called. The address of the array v "allocated" by 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 decremented 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() 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.	10 5 0xffe8 7 1
b.	100 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.	1, 5, 7, 10, 0xffe8
b.	1, 5, 7, 100, 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 from the (luckily) unmodified 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=5, v=7 So v is a 5-element array of integers that begins at address 7
b.	n=5, v=7 So v is a 5-element array of integers that begins at address 7

If we were to actually attempt to access (e.g., print out) elements of array `v` in the `main()` function after this point, the first 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).

3 Solutions

Solution 3.1

3.1.1

a.	3716
b.	6041

3.1.2

a.	3716
b.	1467

3.1.3

a.	1660	1660
b.	2165	–117

3.1.4

a.	6374
b.	753

3.1.5

a.	7504 (–3504)
b.	7777 (–3777)

3.1.6

a.	111000100000
b.	100011110101

The attraction is that each octal digit contains one of 8 different characters (0–7). Since with 3 binary bits you can represent 8 different patterns, in octal each digit requires exactly 3 binary bits. You can write down the conversion directly.

Solution 3.2

3.2.1

a.	7B75
b.	6D95

3.2.2

a.	7B75
b.	6D95

3.2.3

a.	5190	5190
b.	9312	9312

3.2.4

a.	8CA4
b.	5730

3.2.5

a.	FA00
b.	5730

3.2.6

a.	1100001101010010
b.	0101111011010100

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

Solution 3.3

3.3.1

a.	Underflow (–39)
b.	Neither (63)

3.3.2

a.	Overflow (result = –215, which does not fit into an SM 8-bit format)
b.	Neither (65)

3.3.3

a.	Neither (39)
b.	Overflow (result = –179, which does not fit into an SM 8-bit format)

3.3.4

a.	15 – 117 = –102
b.	–105 – 42 = –128 (–147)

3.3.5

a.	15 + 117 = 127 (132)
b.	–105 + 42 = –63

3.3.6

a.	15 + 139 = 154
b.	151 + 214 = 255 (365)

Solution 3.4

3.4.1

a. 62×12

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

b. 35×26

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

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

3.4.2

a. 62×12

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

b. 35×26

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

3.4.3 No solution provided

3.4.4

a. $41 \times 33 = 4033$

Step	Action	Mplier	Multiplicand	Product	Sign
0	Initial Values	011 011	000 000 100 001	000 000 000 000	0
	Multiplier.sign XOR Multiplicand.sign (0 XOR 1)				1
	Make positive	011 011	000 000 000 001	000 000 000 000	1
1	Prod = Prod + Mcand	011 011	000 000 000 001	000 000 000 001	1
	Lshift Mcand	011 011	000 000 000 010	000 000 000 001	1
	Rshift Mplier	001 101	000 000 000 010	000 000 000 001	1
2	Prod = Prod + Mcand	001 101	000 000 000 010	000 000 000 011	1
	Lshift Mcand	001 101	000 000 000 100	000 000 000 011	1
	Rshift Mplier	000 110	000 000 000 100	000 000 000 011	1
3	lsb = 0, no op	000 110	000 000 000 100	000 000 000 011	1
	Lshift Mcand	000 110	000 000 001 000	000 000 000 011	1
	Rshift Mplier	000 011	000 000 001 000	000 000 000 011	1

Step	Action	Mplier	Multiplicand	Product	Sign
4	Prod = Prod + Mcand	000 011	000 000 001 000	000 000 001 011	1
	Lshift Mcand	000 011	000 000 010 000	000 000 001 011	1
	Rshift Mplier	000 001	000 000 010 000	000 000 001 011	1
5	Prod = Prod + Mcand	000 001	000 000 010 000	000 000 011 011	1
	Lshift Mcand	000 001	000 000 100 000	000 000 011 011	1
	Rshift Mplier	000 000	000 000 100 000	000 000 011 011	1
6	lsb = 0, no op	000 000	000 000 100 000	000 000 011 011	1
	Lshift Mcand	000 000	000 001 000 000	000 000 011 011	1
	Rshift Mplier	000 000	000 001 000 000	000 000 011 011	1
7	Prod msb = sign	000 000	000 001 000 000	100 000 011 011	1

b. $60 \times 26 = 4540$

Step	Action	Mplier	Multiplicand	Product	Sign
0	Initial Values	010 110	000 000 110 000	000 000 000 000	0
	Multiplier.sign XOR Multiplicand.sign (0 XOR 1)				1
	Make positive	010 110	000 000 010 000	000 000 000 000	1
1	lsb = 0, no op	010 110	000 000 010 000	000 000 000 000	1
	Lshift Mcand	010 110	000 000 100 000	000 000 000 000	1
	Rshift Mplier	001 011	000 000 100 000	000 000 000 000	1
2	Prod = Prod + Mcand	001 011	000 000 100 000	000 000 100 000	1
	Lshift Mcand	001 011	000 001 000 000	000 000 100 000	1
	Rshift Mplier	000 101	000 001 000 000	000 000 100 000	1
3	Prod = Prod + Mcand	000 101	000 001 000 000	000 001 100 000	1
	Lshift Mcand	000 101	000 010 000 000	000 001 100 000	1
	Rshift Mplier	000 010	000 010 000 000	000 001 100 000	1
4	lsb = 0, no op	000 010	000 010 000 000	000 001 100 000	1
	Lshift Mcand	000 010	000 100 000 000	000 001 100 000	1
	Rshift Mplier	000 001	000 100 000 000	000 001 100 000	1
5	Prod = Prod + Mcand	000 001	000 100 000 000	000 101 100 000	1
	Lshift Mcand	000 001	001 000 000 000	000 101 100 000	1
	Rshift Mplier	000 000	001 000 000 000	000 101 100 000	1

Step	Action	Mplier	Multiplicand	Product	Sign
6	lsb = 0, no op	000 000	001 000 000 000	000 101 100 000	1
	Lshift Mcand	000 000	010 000 000 000	000 101 100 000	1
	Rshift Mplier	000 000	010 000 000 000	000 101 100 000	1
7	Prod msb = sign	000 000	010 000 000 000	100 101 100 000	1

3.4.5

a. $41 \times 33 = -37 \times 33 = -1505$ (6273)

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

b. $60 \times 26 = -20 \times 26 = -540$ (7240)

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

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

3.4.6 No solution provided

Solution 3.5

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

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

a.	$(3 \times 8) \times 4$tu = 96 time units for hardware $(5 \times 8) \times 4$tu = 160 time units for software
b.	$(3 \times 64) \times 8$tu = 1536 time units for hardware $(5 \times 64) \times 8$tu = 2560 time units for software

3.5.2 It takes B time units to get through an adder, and there will be A – 1 adders.

a.	Word is 8 bits wide, requiring 7 adders. 7×4tu = 28 time units.
b.	Word is 64 bits wide, requiring 63 adders. 63×8tu = 504 time units.

3.5.3 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.

a.	8-bit wide word requires 7 adders in 3 levels. 3×4tu = 12 time units.
b.	64-bit word requires 63 adders in 6 levels. 6×8tu = 48 time units.

Solution 3.6

3.6.1

a.	0x33 × 0x55 = 0x10EF. 0x33 = 51, and 51 = 32 + 16 + 2 + 1. We can shift 0x55 left 5 places (0xAA0), then add 0x55 shifted left 4 places (0x550), then add 0x55 shifted left once (0xAA), then add 0x55. 0xAA0 + 0x550 + 0xAA + 0x55 = 0x10EF. 3 shifts, 3 adds. (Could also use 0x55, which is 64 + 16 + 4 + 1, and shift 0x33 left 6 times, add to it 0x33 shifted left 4 times, add to that 0x33 shifted left 2 times, and add to that 0x33. Same number of shifts and adds.)

b.	0x8A × 0xED = 0x7FC2 0x8A = 128 + 8 + 2, 0xED = 128 + 64 + 32 + 8 + 4 + 1. Best way is to shift 0xED left 7 places (0x7680), then add to that 0xED shifted left 3 places (0x768), and then add 0xED shifted left 1 place (0x1DA). 3 shifts, 2 adds.

3.6.2

a.	0x33 × 0x55 = 0x10EF. 0x33 = 51, and 51 = 32 + 16 + 2 + 1. We can shift 0x55 left 5 places (0xAA0), then add 0x55 shifted left 4 places (0x550), then add 0x55 shifted left once (0xAA), then add 0x55. 0xAA0 + 0x550 + 0xAA + 0x55 = 0x10EF. 3 shifts, 3 adds. (Could also use 0x55, which is 64 + 16 + 4 + 1, and shift 0x33 left 6 times, add to it 0x33 shifted left 4 times, add to that 0x33 shifted left 2 times, and add to that 0x33. Same number of shifts and adds.)
b.	0x8A × 0xED = –0x0A × –0x6D = 0x442 0x0A = 8 + 2, 0x6D = 64 + 32 + 8 + 4 + 1. Best way is to shift 0x6D left 3 places (0x368), then add to that 0x6D shifted left 1 place (0xDA). 2 shifts, 1 add.

3.6.3 No solution provided

3.6.4 Quoting the Wikipedia entry directly:

Booth's algorithm involves repeatedly adding one of two predetermined values A and S to a product P, then performing a rightward arithmetic shift on P. Let x and y be the multiplicand and multiplier, respectively; and let x and y represent the number of bits in x and y.

1. Determine the values of A and S, and the initial value of P. All of these numbers should have a length equal to $(x + y + 1)$.
   a. A: Fill the most significant (leftmost) bits with the value of x. Fill the remaining $(y + 1)$ bits with zeros.
   b. S: Fill the most significant bits with the value of $(-x)$ in two's complement notation. Fill the remaining $(y + 1)$ bits with zeros.
   c. P: Fill the most significant x bits with zeros. To the right of this, append the value of y. Fill the least significant (rightmost) bit with a zero.
2. Determine the two least significant (rightmost) bits of P.
   a. If they are 01, find the value of $P + A$. Ignore any overflow.
   b. If they are 10, find the value of $P + S$. Ignore any overflow.
   c. If they are 00 or 11, do nothing. Use P directly in the next step.
3. Arithmetically shift the value obtained in the previous step by a single place to the right. Let P now equal this new value.
4. Repeat steps 2 and 3 until they have been done y times.
5. Drop the least significant (rightmost) bit from P. This is the product of x and y.

3.6.5

a. $0xF6 \times 0x7F = -0xA \times 0x7F = -10 \times 127 = -1270 = 0xFB0A$

Action	Multiplicand	Product/Multiplier
Initial Vals	1111 0110	0000 0000 0111 1111 0
10, subtract shift	1111 0110 1111 0110	0000 1010 0111 1111 0 0000 0101 0011 1111 1
11, nop shift	1111 0110 1111 0110	0000 0101 0011 1111 1 0000 0010 1001 1111 1
11, nop shift	1111 0110 1111 0110	0000 0010 1001 1111 1 0000 0001 0100 1111 1
11, nop shift	1111 0110 1111 0110	0000 0001 0100 1111 1 0000 0000 1010 0111 1
11, nop shift	1111 0110 1111 0110	0000 0000 1010 0111 1 0000 0000 0101 0011 1
11, nop shift	1111 0110 1111 0110	0000 0000 0101 0011 1 0000 0000 0010 1001 1
11, nop shift	1111 0110 1111 0110	0000 0000 0010 1001 1 0000 0000 0001 0100 1
01, add shift	1111 0110 1111 0110	1111 0110 0001 0100 1 1111 1011 0000 1010 0

b. $0x08 \times 0x55 = 0x2A8$

Action	Multiplicand	Product/Multiplier
Initial Vals	0000 1000	0000 0000 0101 0101 0
10, subtract shift	0000 1000 0000 1000	1111 1000 0101 0101 0 1111 1100 0010 1010 1
01, add shift	0000 1000 0000 1000	0000 0100 0010 1010 1 0000 0010 0001 0101 0
10, subtract shift	0000 1000 0000 1000	1111 1010 0001 0101 0 1111 1101 0000 1010 1
01, add shift	0000 1000 0000 1000	0000 0101 0000 1010 1 0000 0010 1000 0101 0
10, subtract shift	0000 1000 0000 1000	1111 1010 1000 0101 0 1111 1101 0100 0010 1
01, add shift	0000 1000 0000 1000	0000 0101 0100 0010 1 0000 0010 1010 0001 1

Action	Multiplicand	Product/Multiplier
10, subtract shift	0000 1000 0000 1000	1111 1010 1010 0001 0 1111 1101 0101 0000 1
01, add shift	0000 1000 0000 1000	0000 0101 0101 0000 1 0000 0010 1010 1000 1

3.6.6 No solution provided

Solution 3.7

3.7.1

a. $74/21 = 3$ 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

b. 76/52 = 1 remainder 24

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

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

a. 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

Step	Action	Divisor	Remainder/Quotient
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
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

b. $76/52 = 1$ remainder 24

Step	Action	Divisor	Remainder/Quotient
0	Initial Vals	101 010	000 000 111 110
1	R<<	101 010	000 001 111 100
	Rem = Rem – Div	101 010	101 001 111 100
	Rem < 0, R + D	101 010	000 001 111 100
2	R<<	101 010	000 011 111 000
	Rem = Rem – Div	101 010	100 111 111 000
	Rem < 0, R + D	101 010	000 011 111 000
3	R<<	101 010	000 111 110 000
	Rem = Rem – Div	101 010	100 011 110 000
	Rem < 0, R + D	101 010	000 111 110 000
4	R<<	101 010	001 111 100 000
	Rem = Rem – Div	101 010	100 101 100 000
	Rem < 0, R + D	101 010	001 111 100 000

Step	Action	Divisor	Remainder/Quotient
5	R<<	101 010	011 111 000 000
	Rem = Rem – Div	101 010	110 101 000 000
	Rem < 0, R + D	101 010	011 111 000 000
6	R<<	101 010	111 110 000 000
	Rem = Rem – Div	101 010	010 100 000 000
	Rem > 0, R0 = 1	101 010	010 100 000 001

3.7.3 No solution provided

3.7.4

a. 72/07 = 3 remainder 5: Dividend negative

Sign of Quotient = (Sign bit of Divisor) XOR (Sign bit of Dividend) = negative
Sign of Remainder = Sign of Dividend = negative

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

Step	Action	Quotient	Divisor	Remainder
7	Rem = Rem – Div	000 001	000 000 000 111	000 000 000 101
	Rem < 0, Q << 1	000 011	000 000 000 111	000 000 000 101
	Rshift Div	000 011	000 000 000 011	000 000 000 101
8	Set sign bits	100 011	000 000 000 011	100 000 000 101

b. 75/44 = 7 remainder 1: Dividend negative

Sign of Quotient = (Sign bit of Divisor) XOR (Sign bit of Dividend) = positive
Sign of Remainder = Sign of Dividend = negative

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

3.7.5

a. 72/07 = 3 remainder 5: Dividend negative

Sign of Quotient = (Sign bit of Divisor) XOR (Sign bit of Dividend) = negative
Sign of Remainder = Sign of Dividend = negative

Step	Action	Divisor	Remainder/Quotient
0	Initial Vals	000 111	000 000 011 010
1	R<<	000 111	000 000 110 100
	Rem = Rem – Div	000 111	111 001 110 100
	Rem < 0, R + D	000 111	000 000 110 100
2	R<<	000 111	000 001 101 000
	Rem = Rem – Div	000 111	111 010 101 000
	Rem < 0, R + D	000 111	000 001 101 000
3	R<<	000 111	000 011 010 000
	Rem = Rem – Div	000 111	111 100 010 000
	Rem < 0, R + D	000 111	000 011 010 000
4	R<<	000 111	000 110 100 000
	Rem = Rem – Div	000 111	111 111 100 000
	Rem < 0, R + D	000 111	000 110 100 000
5	R<<	000 111	001 101 000 000
	Rem = Rem – Div	000 111	000 110 000 000
	Rem > 0, R0 = 1	000 111	000 110 000 001
6	R<<	000 111	001 100 000 010
	Rem = Rem – Div	000 111	000 101 000 010
	Rem > 0, R0 = 1	000 111	000 101 000 011
7	Adjust signs	000 111	100 101 100 011 (Q = –3, Rem = –5)

b. 75/44 = 7 remainder 1: Dividend negative

Sign of Quotient = (Sign bit of Divisor) XOR (Sign bit of Dividend) = positive
Sign of Remainder = Sign of Dividend = negative

Step	Action	Divisor	Remainder/Quotient
0	Initial Vals	000 100	000 000 011 101
1	R<<	000 100	000 000 111 010
	Rem = Rem – Div	000 100	111 100 111 010
	Rem < 0, R + D	000 100	000 000 111 010

Step	Action	Divisor	Remainder/Quotient
2	R<<	000 100	000 001 110 100
	Rem = Rem – Div	000 100	111 101 110 100
	Rem < 0, R + D	000 100	000 001 110 100
3	R<<	000 100	000 011 101 000
	Rem = Rem – Div	000 100	111 111 101 000
	Rem < 0, R + D	000 100	000 011 101 000
4	R<<	000 100	000 111 010 000
	Rem = Rem – Div	000 100	000 011 010 000
	Rem > 0, R0 = 1	000 100	000 011 010 001
5	R<<	000 100	000 110 100 010
	Rem = Rem – Div	000 100	000 010 100 010
	Rem > 0,R0 = 1	000 100	000 010 100 011
6	R<<	000 100	000 101 000 110
	Rem = Rem – Div	000 100	000 001 000 110
	Rem > 0, R0 = 1	000 100	000 001 000 111
7	Adjust signs	000 100	100 001 000 111 (Q = 7, Rem = –1)

3.7.6 No solution provided

Solution 3.8

3.8.1 In these solutions a 1 will be shifted into the quotient and a compensating right shift of the remainder will be performed. This is the alternate approach mentioned in Solution Solution 3.7.2: In these solutions a 1 or a 0 was added to the quotient if the remainder was greater than or equal to 0..

a. 26/05 = 4 remainder 2

Step	Action	Divisor	Remainder/Quotient
0	Initial Vals	000 101	000 000 010 110
	R<<	000 101	000 000 101 100
	Rem = Rem – Div	000 101	111 011 101 100
1	Rem < 0, Q << 0, Addnext	000 101	110 111 011 000
	Rem = Rem + Div	000 101	111 100 011 000
2	Rem < 0, Q << 0, Addnext	000 101	111 000 110 000
	Rem = Rem + Div	000 101	111 101 110 000

Step	Action	Divisor	Remainder/Quotient
3	Rem < 0, Q << 0, Addnext	000 101	111 011 100 000
	Rem = Rem + Div	000 101	000 000 100 000
4	Rem > = 0, Q << 1, Sub	000 101	000 001 000 001
	Rem = Rem – Div	000 101	111 100 000 001
5	Rem < 0, Q << 0, Add	000 101	111 000 000 010
	Rem = Rem + Div	000 101	111 101 000 010
6	Rem < 0, Q << 0, Add	000 101	111 010 000 100
	Rem = Rem + Div	000 101	111 111 000 100
7	Rem < 0, Rem = Rem + Div	000 101	000 100 000 100
	Shift Rem >> 1	000 101	000 010 000 100 (Q = 4, Rem = 2)

b. 37/15 = 2 remainder 5

Step	Action	Divisor	Remainder/Quotient
0	Initial Vals	001 101	000 000 011 111
	R<<	001 101	000 000 111 110
	Rem = Rem – Div	001 101	110 011 111 110
1	Rem < 0, Q << 0, Addnext	001 101	100 111 111 100
	Rem = Rem + Div	001 101	110 100 111 100
2	Rem < 0, Q << 0, Addnext	001 101	101 001 111 000
	Rem = Rem + Div	001 101	110 110 111 000
3	Rem < 0, Q << 0, Addnext	001 101	101 101 110 000
	Rem = Rem + Div	001 101	111 010 110 000
4	Rem < 0, Q << 0, Addnext	001 101	110 101 100 000
	Rem = Rem + Div	001 101	000 010 100 000
5	Rem > 0, Q << 1, Subnext	001 101	000 101 000 001
	Rem = Rem – Div	001 101	111 000 000 001
6	Rem < 0, Q << 0, Addnext	001 101	110 000 000 010
	Rem = Rem + Div	001 101	111 101 000 010
7	Rem < 0, Rem = Rem + Div	001 101	001 010 000 010
	Shift Rem >> 1	001 101	000 101 000 010 (Q = 2, Rem = 5)

3.8.2 No solution provided

3.8.3 No solution provided

3.8.4

a. 27/6 = 3 remainder 5

Step	Action	Quotient	Temp	Divisor	Remainder
0	Initial Vals	000000	000000 000000	000110 000000	000000 010111
1	Temp = Rem – Div	000000	111010 010111	000110 000000	000000 010111
	Temp < 0, Q << 0	000000	111010 010111	000110 000000	000000 010111
	Rshift Div	000000	111010 010111	000011 000000	000000 010111
2	Temp = Rem – Div	000000	111101 010111	000011 000000	000000 010111
	Temp < 0, Q << 0	000000	111101 010111	000011 000000	000000 010111
	Rshift Div	000000	111101 010111	000001 100000	000000 010111
3	Temp = Rem – Div	000000	111111 110111	000001 100000	000000 010111
	Temp < 0, Q << 0	000000	111111 110111	000001 100000	000000 010111
	Rshift Div	000000	111111 110111	000000 110000	000000 010111
4	Temp = Rem – Div	000000	111111 100111	000000 110000	000000 010111
	Temp < 0, Q << 0	000000	111111 100111	000000 110000	000000 010111
	Rshift Div	000000	111111 100111	000000 011000	000000 010111
5	Temp = Rem – Div	000000	111111 111111	000000 011000	000000 010111
	Temp < 0, Q << 0	000000	111111 111111	000000 011000	000000 010111
	Rshift Div	000000	111111 111111	000000 001100	000000 010111
6	Temp = Rem – Div	000000	000000 001011	000000 001100	000000 010111
	T > 0, Q << 1, R = T	000001	000000 001011	000000 001100	000000 001011
	Rshift Div	000001	000000 001011	000000 000110	000000 001011
7	Temp = Rem – Div	000001	000000 000101	000000 000110	000000 001011
	T > 0, Q << 1, R = T	000011	000000 000101	000000 000110	000000 000101
	Rshift Div	000011	000000 000101	000000 000011	000000 000101

b. 54/12 = 4 remainder 4

Step	Action	Quotient	Temp	Divisor	Remainder
0	Initial Vals	000000	000000 000000	001010 000000	000000 101100
1	Temp = Rem – Div	000000	110110 101100	001010 000000	000000 101100
	Temp < 0, Q << 0	000000	110110 101100	001010 000000	000000 101100
	Rshift Div	000000	110110 101100	000101 000000	000000 101100
2	Temp = Rem – Div	000000	111011 101100	000101 000000	000000 101100
	Temp < 0, Q << 0	000000	111011 101100	000101 000000	000000 101100
	Rshift Div	000000	111011 101100	000010 100000	000000 101100

Step	Action	Quotient	Temp	Divisor	Remainder
3	Temp = Rem – Div	000000	111110 001100	000010 100000	000000 101100
	Temp < 0, Q << 0	000000	111110 001100	000010 100000	000000 101100
	Rshift Div	000000	111110 001100	000001 010000	000000 101100
4	Temp = Rem – Div	000000	111111 011100	000001 010000	000000 101100
	Temp < 0, Q << 0	000000	111111 011100	000001 010000	000000 101100
	Rshift Div	000000	111111 011100	000000 101000	000000 101100
5	Temp = Rem – Div	000000	000000 000100	000000 101000	000000 101100
	T > 0, Q << 1, R = T	000001	000000 000100	000000 101000	000000 000100
	Rshift Div	000001	000000 000100	000000 010100	000000 000100
6	Temp = Rem – Div	000001	111111 110000	000000 010100	000000 000100
	Temp < 0, Q << 0	000010	111111 110000	000000 010100	000000 000100
	Rshift Div	000010	111111 110000	000000 001010	000000 000100
7	Temp = Rem – Div	000010	111111 111010	000000 001010	000000 000100
	Temp < 0, Q << 0	000100	111111 111010	000000 001010	000000 000100
	Rshift Div	000100	111111 111010	000000 000101	000000 000100

3.8.5 No solution provided

3.8.6 No solution provided

Solution 3.9

3.9.1 No solution provided

3.9.2 No solution provided

3.9.3 No solution provided

Solution 3.10

3.10.1

a.	201326592	201326592
b.	–1000144896	3294822400

3.10.2

a.	`jal 0x00000000`
b.	`lwc1 $3,0($3)`

3.10.3

a.	0x0C000000 = 0000 1100 0000 0000 0000 0000 0000 0000 = 0 0001 1000 0000 0000 0000 0000 0000 000 sign is positive exp = 0x18 = 24 – 127 = –103 there is a hidden 1 mantissa = 0 answer = 1.0 x 2^{-103}
b.	0xC4630000 = 1100 0100 0110 0011 0000 0000 0000 0000 = 1 1000 1000 1100 0110 0000 0000 0000 000 sign is negative exp = 0x88 = 136 – 127 = 9 there is a hidden 1 mantissa = 0xC60000 = 12 x 16^{-1} + 6 x 16^{-2} = .75 +.0234375 answer = –1.7734375 x 2^{9}

3.10.4

a.	63.25 x 10^{0} = 111111.01 x 2^{0} normalize, move binary point 5 to the left 1.1111101 x 2^{5} sign = positive, exp = 127 + 5 = 132 Final bit pattern: 0 1000 0100 1111 1010 0000 0000 0000 000 = 0100 0010 0111 1101 0000 0000 0000 0000 = 0x427D0000
b.	146987.40625 x 10^{0} = 100011111000101011.011010 x 2^{0} normalize, move binary point 17 to the left 1.00011111000101011011010 x 2^{17} sign = positive, exp = 127 + 17 = 144 Final bit pattern: 0 1001 0000 0001 1111 0001 0101 1011 010 = 0100 1000 0000 1111 1000 1010 1101 1010 = 0x480F8ADA

3.10.5

a.	63.25 x 10^{0} = 111111.01 x 2^{0} normalize, move binary point 5 to the left 1.1111101 x 2^{5} sign = positive, exp = 1023 + 5 = 1028 Final bit pattern: 0 100 0000 0100 1111 1010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 = 0x404FA00000000000
b.	146987.40625 x 10^{0} = 100011111000101011.011010 x 2^{0} normalize, move binary point 17 to the left 1.00011111000101011011010 x 2^{17} sign = positive, exp = 1023 + 17 = 1040 Final bit pattern: 0 100 0001 0000 0001 1111 0001 0101 1011 0100 0000 0000 0000 0000 0000 0000 0000 = 0x4101F15B40000000

3.10.6

a.	$63.25 \times 10^0 = 111111.01 \times 2^0 = 3F.40 \times 16^0$ move hex point 2 to the left $.3F40 \times 16^2$ sign = positive, exp = 64 + 2 Final bit pattern: 01000010001111110100000000000000
b.	$146987.40625 \times 10^0 = 10\ 0011\ 1110\ 0010\ 1011.011010 \times 2^0$ $= 23E2B.68 \times 16^0$ move hex point 5 to the left $.00100011111000101011011010 \times 16^5$ sign = positive, exp = 64 + 5 = 69 Final bit pattern: 01000101001000111110001010110110

Solution 3.11

3.11.1

a.	$-1.5625 \times 10^{-1} = -.15625 \times 10^0$ $= -.00101 \times 2^0$ move the binary point 2 to the right $= -.101 \times 2^{-2}$ exponent = –2, mantissa = –.1010000000000000000000 answer: 111111111110101100000000000000000000
b.	$9.356875 \times 10^2 = 935.6875 \times 10^0$ $= 0x3A7.B \times 16^0 = 1110100111.1011 \times 2^0$ move the binary point 10 to the left $= .11101001111011 \times 2^{10}$ exponent = +10, mantissa = +.11101001111011 answer: 000000001010011101001111011000000000

3.11.2

a.	$-1.5625 \times 10^{-1} = -.15625 \times 10^0$ $= -.00101 \times 2^0$ move the binary point 3 to the right, $= -1.01 \times 2^{-3}$ exponent = –3 = –3 + 16 = 13, mantissa = –.0100000000 answer: 1011010100000000
b.	$9.356875 \times 10^2 = 935.6875 \times 10^0$ $= 0x3A7.B \times 16^0 = 1110100111.1011 \times 2^0$ move the binary point 9 to the left $= 1.1101001111011 \times 2^9$ exponent = +9 = 9 + 16 = 25, mantissa = +.1101001111011 answer: 0110011101001111

3.11.3

a.	$-1.5625 \times 10^{-1} = -.15625 \times 10^{0}$ $= -.00101 \times 2^{0}$ move the binary point 2 to the right $= -.101 \times 2^{-2}$ exponent = –2, mantissa = –.10100000000000000000000000 answer: 10110000000000000000000000000101
b.	$9.356875 \times 10^{2} = 935.6875 \times 10^{0}$ $= 0x3A7.B \times 16^{0} = 1110100111.1011 \times 2^{0}$ move the binary point 10 to the left $= .11101001111011 \times 2^{10}$ exponent = +10, mantissa = +.11101001111011 answer: 01110100111101100000000000010100

3.11.4

a.	$2.6125 \times 10^{1} + 4.150390625 \times 10^{-1}$
	$2.6125 \times 10^{1} = 26.125 = 11010.001 = 1.1010001000 \times 2^{4}$
	$4.150390625 \times 10^{-1} = .4150390625 = .011010100111 = 1.1010100111 \times 2^{-2}$
	Shift binary point 6 to the left to align exponents,
	`              GR`
	`1.1010001000 00`
	`+.0000011010 10 0111  (Guard = 1, Round = 0, Sticky = 1)`
	`--------------------`
	`1.1010100010 10`
	In this case the extra bits (G,R,S) are more than half of the least significant bit (0).
	Thus, the value is rounded up.
	$1.1010100011 \times 2^{4} = 11010.100011 \times 2^{0} = 26.546875 = 2.6546875 \times 10^{1}$

b.	$-4.484375 \times 10^{1} + 1.3953125 \times 10^{1}$
	$-4.484375 \times 10^{1} = -44.84375 = -1.0110011011 \times 2^{5}$
	$1.1953125 \times 10^{1} = 11.953125 = 1.0111111010 \times 2^{3}$
	Shift binary point 2 to the left and align exponents,
	`              GR`
	`-1.0110011011 00`
	` 0.0101111110 10   (Guard = 1, Round = 0, Sticky = 0)`
	`------------------`
	`-1.0000011100 10`
	In this case, the Guard is 1 and the Round and Sticky bits are zero. This is the "exactly half" case—if the LSB was odd (1) we would add, but since it is even (0) we do nothing.
	$-1.0000011100 \times 2^{5} = -100000.11100 \times 2^{0} = -32.875 = -3.2875 \times 10^{1}$

3.11.5 No solution provided

3.11.6 No solution provided

Solution 3.12

3.12.1

a.

$-8.0546875 \times -1.79931640625 \times 10^{-1}$
$-8.0546875 = -1.0000000111 \times 2^{3}$
$-1.79931640625 \times 10^{-1} = -1.0111000010 \times 2^{-3}$

Exp: $-3 + 3 = 0$, $0 + 16 = 16$ (10000)
Signs: both negative, result positive

Mantissa:

```
          1.0000000111
        × 1.0111000010
          ------------
           00000000000
          10000000111
         00000000000
        00000000000
       00000000000
      00000000000
     10000000111
    10000000111
   10000000111
  00000000000
 10000000111
1.01110011000001001110
```

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

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

$-8.0546875 \times -.179931640625 = 1.4492931365966796875$

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

b.

$8.59375 \times 10^{-2} \times 8.125 \times 10^{-1}$
$8.59375 \times 10^{-2} = .0859375 = 1.0110000000 \times 2^{-4}$
$8.125 \times 10^{-1} = .8125 = 1.1010000000 \times 2^{-1}$

Exp: $-4 + -1 = -5$, $-5 + 16 = 11$ (01011)
Signs: both positive, result positive

Mantissa:

```
          1.0110000000
        × 1.1010000000
          ------------
           00000000000
          00000000000
         00000000000
        00000000000
       00000000000
      00000000000
     00000000000
    10110000000
   00000000000
  10110000000
 10110000000
10001111000000000000000 Normalize, add one to exponent, negate
```

–1.0001111000 00 000000000 Guard = 0, Round = 0, Sticky = 0: Nornd

$-1.0001111000 \times 2^{-4}$ = 1011000001111000 (.00010001111000) = .06982421875

.0859375 × .8125 = .06982421875

In this case the two match exactly, since no information was lost during the shifting.

3.12.2 No solution provided

3.12.3 No solution provided

3.12.4

a.

$8.625 \times 10^1 / -4.875 \times 10^0$
$8.625 \times 10^1 = 1.0101100100 \times 2^6$
$-4.875 = -1.0011100000 \times 2^2$

Exponent = 6 – 2 = 4, 4 + 16 = 20 (10100)
Signs: one positive, one negative, result negative

Mantissa:

```
                                  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
                                   ------------------
                                   .00000000011000000
```

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

-1.0001101100×2^4 = 1101000001101100 = 10001.101100 = –17.6875

86.25 / –4.875 = –17.692307692307

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

b. $1.84375 \times 10^0 / 1.3203125 \times 10^0$
$1.84375 \times 10^0 = 1.84375 = 1.1101100000 \times 2^0$
$1.3203125 \times 10^0 = 1.3203125 = 1.0101001000 \times 2^0$

Exponent = 0–0 = 0, 0 + 16 = 16 (10000)
Signs: both positive, result positive

Mantissa:

```
                         1.011001010111110
10101001000. | 11101100000.0000000000000000
              -10101001000.
              ------------
                1000011000.00
               - 101010010.00
                 ------------
                  11000110.000
                - 10101001.000
                  ------------
                     11101.000000
                   - 10101.001000
                     ------------
                       111.11100000
                     - 101.01001000
                       ------------
                        10.1001100000
                      -  1.0101001000
                         ------------
                         1.01000110000
                       -  .10101001000
                          ------------
                          .100111010000
                       -  .010101001000
                           ------------
                          .0100100010000
                       -  .0010101001000
                           ------------
                          .00011110010000
                       -  .00010101001000
                           ------------
                          .00001001001000
                       -  .000010101001000
```

1.0110010101 11 110 Guard = 1, Round = 1, Sticky = 1: Round up

1.0110010110×2^0 = 0100000110010110 = 1.0110010110 = 1.396484375

1.84375 / 1.3203125 = 1.39644970414201183431952 66

Some information was lost because the result did not fit into the available 10-bit field. Answer off by .000034671.

3.12.5 No solution provided

3.12.6 No solution provided

Solution 3.13

3.13.1

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

$(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^{3})$
$3.984375 \times 10^{-1} = 1.1001100000 \times 2^{-2}$
$3.4375 \times 10^{-1} = 1.0110000000 \times 2^{-2}$
$1.771 \times 10^{3} = 1771 = 1.1011101011 \times 2^{10}$
shift binary point of smaller left 12 so exponents match

```
(A)      1.1001100000
(B)     +1.0110000000
       --------------
        10.1111100000 Normalize,
(A+B)    1.0111110000 × 2^-1
(C)     +1.1011101011
(A+B)     .0000000000 10 111110000 Guard=1, Round=0, Sticky=1
       --------------
(A+B)+C +1.1011101011 10 1 Round up
(A+B)+C =1.1011101100 × 2^10 = 0110101011101100 = 1772
```

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

$(3.96875 \times 10^{0} + 8.46875 \times 10^{0}) + 2.1921875 \times 10^{1}$
$3.96875 \times 10^{0} = 1.1111110000 \times 2^{1}$
$8.46875 \times 10^{0} = 1.0000111100 \times 2^{3}$
$2.1921875 \times 10^{1} = 1.0101111011 \times 2^{4}$
shift binary point of smaller left 6 so exponents match

```
(A)        .0111111100 00 Guard=0, Round=0, Sticky=0
(B)       1.0000111100
        --------------
(A+B)     1.1000111000 No round
(A+B)      .1100011100 0  Guard=0, Round=0, Sticky=0
(C)      +1.0101111011
        --------------
(A+B)+C  10.0010010111 Normalize, add 1 to exponent, round to even
(A+B)+C = 1.0001001100 × 2^5 = 0101010001001100 = 34.375
```

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

3.13.2

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

$3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^{3})$
$3.984375 \times 10^{-1} = 1.1001100000 \times 2^{-2}$
$3.4375 \times 10^{-1} = 1.0110000000 \times 2^{-2}$
$1.771 \times 10^{3} = 1771 = 1.1011101011 \times 2^{10}$
shift binary point of smaller left 12 so exponents match

```
(B)       .0000000000  01 0110000000 Guard=0, Round=1, Sticky=1
(C)     +1.1011101011
       --------------
(B+C)   +1.1011101011
(A)       .0000000000 011001100000
       --------------
A+(B+C) +1.1011101011 No round
A+(B+C) +1.1011101011 × 2^10 = 0110101011101011 = 1771
```

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

b.	$3.96875 \times 10^0 + (8.46875 \times 10^0 + 2.1921875 \times 10^1)$
	$3.96875 \times 10^0 = 1.1111110000 \times 2^1$
	$8.46875 \times 10^0 = 1.0000111100 \times 2^3$
	$2.1921875 \times 10^1 = 1.0101111011 \times 2^4$
	shift binary point of smaller left 6 so exponents match
	`(B)        .1000011110 0 Guard=0, Round=0, Sticky=0`
	`(C)       1.0101111011`
	`        --------------`
	`(B+C)     1.1110011001 No round`
	`(A)        .0011111110 000 Guard=0, Round=0, Sticky=0`
	`(B+C)     1.1110011001`
	`        --------------`
	`(A+B)+C  10.0010010111 Normalize, add 1 to exponent, round to even`
	(A+B)+C = 1.0001001100 × 2^5 = 0101010001001100 = 34.375

3.13.3

a.	No, they are not equal: (A + B) + C = 1772, A + (B + C) = 1771 (steps shown above). Exact: .398437 + .34375 + 1771 = 1771.742187.
b.	Yes, they are equal: (A + B) + C = 34.375, A + (B + C) = 34.375 (steps shown above). Exact answer is 34.359375.

3.13.4

a.	$(3.41796875 \times 10^{-3} \times 6.34765625 \times 10^{-3}) \times 1.05625 \times 10^2$
	(A) $3.41796875 \times 10^{-3} = 1.1100000000 \times 2^{-9}$
	(B) $4.150390625 \times 10^{-3} = 1.0001000000 \times 2^{-8}$
	(C) $1.05625 \times 10^2 = 1.1010011010 \times 2^6$
	Exp: –9 –8 = –17
	Signs: both positive, result positive
	Mantissa:
	`(A)                  1.1100000000`
	`(B)                × 1.0001000000`
	`                     ------------`
	`               11100000000`
	`           11100000000`
	`          ----------------------`
	`          1.11011100000000000000`
	`A×B       1.1101110000 00 00000000 Guard = 0, Round = 0, Sticky = 0: No Round`
	A×B 1.1101110000 × 2^{-17} UNDERFLOW: Cannot represent number

b.	$(1.140625 \times 10^2 \times -9.135 \times 10^2) \times 9.84375 \times 10^{-1}$
	(A) $1.140625 \times 10^2 = 1.1100100001 \times 2^6$
	(B) $-9.135 \times 10^2 = -1.1100100011 \times 2^9$
	(C) $9.84375 \times 10^{-1} = 1.1111100000 \times 2^{-1}$
	Exp: 6 + 9 = 15
	Signs: one positive, one negative - result negative
	Mantissa:
	`(A)                      1.1100100001`
	`(B)                    × 1.1100100011`
	`                         ------------`
	`                          11100100001`
	`                         11100100001`
	`                     11100100001`
	`                  11100100001`
	`                 11100100001`
	`                11100100001`
	`               ----------------------`
	`             11.00101110000010000011 Normalize, add 1 to exponent`
	`               1.1001011100 00 010000011 Guard=0, Round=0, Sticky=1: No Round`
	A × B $-1.1001011100 \times 2^{16}$ OVERFLOW: Cannot represent number

3.13.5

a.	$3.41796875 \times 10^{-3} \times (6.34765625 \times 10^{-3} \times 1.05625 \times 10^{2})$
	(A) $3.41796875 \times 10^{-3} = 1.1100000000 \times 2^{-9}$
	(B) $4.150390625 \times 10^{-3} = 1.0001000000 \times 2^{-8}$
	(C) $1.05625 \times 10^{2} = 1.1010011010 \times 2^{6}$
	Exp: $-8 + 6 = -2$
	Signs: both positive, result positive
	Mantissa:
	`(B)                    1.0001000000`
	`(C)                  × 1.1010011010`
	`                       ------------`
	`                       10001000000`
	`                     10001000000`
	`                    10001000000`
	`                 10001000000`
	`               10001000000`
	`              10001000000`
	`             ----------------------`
	`             1.110000001110100000000`
	`             1.1100000011 10 100000000  Guard=1, Round=0, Sticky=1: Round`
	B × C $1.1100000100 \times 2^{-2}$
	Exp: $-9 - 2 = -11$
	Signs: both positive, result positive
	Mantissa:
	`(A)                    1.1100000000`
	`(B × C)              × 1.1100000100`
	`                       ------------`
	`                      11100000000`
	`                11100000000`
	`               11100000000`
	`              11100000000`
	`             ----------------------`
	`            11.00010001110000000000 Normalize, add 1 to exponent`
	`             1.1000100011 10 0000000000 Guard = 1, Round = 0, Sticky = 0: Round to even`
	A × (B × C) $1.1000100100 \times 2^{-10}$

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

$1.140625 \times 10^2 \times (-9.135 \times 10^2 \times 9.84375 \times 10^{-1})$

(A) $1.140625 \times 10^2 = 1.1100100001 \times 2^6$
(B) $-9.135 \times 10^2 = -1.1100100011 \times 2^9$
(C) $9.84375 \times 10^{-1} = 1.1111100000 \times 2^{-1}$

Exp: 9 – 1 = 8
Signs: one negative, one positive - result negative

Mantissa:

<pre>
(B)   1.1100100011
(C)   × 1.1111100000

                   11100100011
                  11100100011
                 11100100011
                11100100011
               11100100011
              11100100011

            11.100000110011101 Normalize, add 1 to exponent
             1.1100000110 01 1101000000 Guard=0, Round=1, Sticky=1:
             No Round
</pre>

B × C -1.1100000110×2^9
Exp: 5 + 9 = 14
Signs: one negative, one positive - result negative

Mantissa:

<pre>
(A)   1.1100100001
(B×C)   × 1.1100000110

                       11100100001
                      11100100001
                11100100001
               11100100001
              11100100001

            11.00100001000111000110 Normalize, add 1 to exponent
             1.1001000010 00 111000110 Guard=0, Round=0, Sticky=1:
             No Round
</pre>

A × (B × C) $1.1001000010 \times 2^{15}$

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

3.13.6

a.	b) No: A × B = $1.1101110000 \times 2^{-17}$ UNDERFLOW: Cannot represent A × (B × C) = $1.1000100100 \times 2^{-10}$ A and B are both small, so their product does not fit into the 16-bit floating point format being used.
b.	e) No: A × (B × C) = $-1.1001000010 \times 2^{15}$ A × B = $-1.1001011100 \times 2^{16}$ OVERFLOW: Cannot be represented A and B are both large, so their product does not fit into the 16-bit floating point format being used.

Solution 3.14

3.14.1

a. $1.666015625 \times 10^0 \times (1.9760 \times 10^4 - 1.9744 \times 10^4)$

(A) $1.666015625 \times 10^0 = 1.1010101010 \times 2^0$
(B) $1.9760 \times 10^4 = 1.0011010011 \times 2^{14}$
(C) $-1.9744 \times 10^4 = -1.0011010010 \times 2^{14}$

Exponents match, no shifting necessary

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

Exp: $0 + 4 = 4$
Signs: both positive, result positive

Mantissa:

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

$A \times (B + C)$ 1.1010101010×2^4

b. $3.48 \times 10^2 \times (6.34765625 \times 10^{-2} - 4.052734375 \times 10^{-2})$

(A) $3.48 \times 10^2 = 1.0101110000 \times 2^8$
(B) $6.34765625 \times 10^{-2} = 1.0000010000 \times 2^{-4}$
(C) $-4.052734375 \times 10^{-2} = 1.0100110000 \times 2^{-5}$

Shift binary point of smaller left 1 so exponents match

```
(B)   1.0000010000 × 2^-4
(C)   -.1010011000 0 × 2^-4
      ---------------
(B+C)  .0101111000 Normalize, subtract 2 from exponent
```

$(B + C)$ $1.0111100000 \times 2^{-6}$
Exp: $8 - 6 = 2$
Signs: both positive, result positive

Mantissa:

```
(A)                        1.0101110000
(B+C)                    × 1.0111100000
                           ------------
                    10101110000
                   10101110000
                  10101110000
                 10101110000
               10101110000
A×(B+C)        1.1111111100 10000000000 Guard=1, Round=0, Sticky=0:
               Round to even
```

$A \times (B + C)$ 1.1111111100×2^2

3.14.2

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

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

(A) $1.666015625 \times 10^0 = 1.1010101010 \times 2^0$
(B) $1.9760 \times 10^4 = 1.0011010011 \times 2^{14}$
(C) $-1.9744 \times 10^4 = -1.0011010010 \times 2^{14}$

Exp: 0 + 14 = 14
Signs: both positive, result positive

Mantissa:

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

Exp: 0 + 14 = 14
Signs: one negative, one positive, result negative

Mantissa:

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

$A \times B + A \times C \;\; 1.0000000000 \times 2^5$

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

b.	$3.48 \times 10^2 \times (6.34765625 \times 10^{-2} - 4.052734375 \times 10^{-2})$
	(A) $3.48 \times 10^2 = 1.0101110000 \times 2^8$
	(B) $6.34765625 \times 10^{-2} = 1.0000010000 \times 2^{-4}$
	(C) $-4.052734375 \times 10^{-2} = 1.0100110000 \times 2^{-5}$
	Exp: 8 – 4 = 4
	Signs: both positive, result positive
	Mantissa:
	`(A)                    1.0101110000`
	`(B)                  × 1.0000010000`
	`                       ------------`
	`                    10101110000`
	`              10101110000`
	`             ----------------------`
	`             1.01100001011100000000`
	`A×B          1.0110000101 11 00000000 Guard=1, Round=1, Sticky=0: Round`
	A × B 1.0110000110×2^4
	Exp: 8 – 5 = 3
	Signs: one negative, one positive, result negative
	Mantissa:
	`(A)                    1.0101110000`
	`(C)                  × 1.0100110000`
	`                       ------------`
	`                    10101110000`
	`                   10101110000`
	`                10101110000`
	`              10101110000`
	`             ------------------------`
	`              1.11000011010100000000`
	`A×C           1.1100001101 0100000000 Guard=0, Round=1, Sticky=0: No Round`
	A × C -1.1100001101×2^3
	`A×B      1.0110000110 × 2^4`
	`A×C      -.1110000110 1 × 2^4 (Guard=1, Round=0, Sticky=0: Round to even)`
	`         -------------`
	`A×B+A×C   .1000000000 × 2^4`
	A × B + A × C 1.000000000×2^3

3.14.3

a.	b) No: $A \times (B + C) = 1.1010101010 \times 2^4 = 26.65625$, and $(A \times B) + (A \times C) = 1.0000000000 \times 2^5 = 32$ Exact: $1.666015625 \times (19760 - 19744) = 26.65625$
b.	e) No: $A \times B + A \times C = 1.0000000000 \times 2^3 = 8$, and $A \times (B + C) = 1.1111111100 \times 2^2 = 7.984375$ Exact: $348 \times (.0634765625 - .04052734375) = 7.986328125$

3.14.4

	Answer	Sign	Exp	Exact?
a.	1 01111101 00000000000000000000000	–	–2	Yes
b.	0 01111011 10011001100110011001101	+	–4	No

3.14.5

a.	b + b + b + b = –1 b × 4 = –1 They are the same
b.	e + e + e + e + e + e + e + e + e + e = 1.000000000000000000000100 e × 10 = 1.000000000000000000000100

3.14.6 No solution provided

Solution 3.15

3.15.1

a.	0101 0101 0101 0101 0101 0101	0x.555555	No
b.	0001 1001 1001 1001 1001 1001	.199999	No

3.15.2

a.	0011 0011 0011 0011 0011 0011	.33333	No
b.	0001 0000 0000 0000 0000 0000	.100000	Yes

3.15.3

a.	0101 0000 0000 0000 0000 0000	.500000	Yes
b.	0001 0111 0111 0111 0111 0111	.177777	No

3.15.4

a.	01010 00000 00000 00000	.A000	Yes
b.	00011 00000 00000 00000	.3000	Yes

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	AND	1 (ALU)	0
b.	0	0	1 (Imm)	1	ADD	X	0

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

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

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

4.1.2 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 write port of the Registers

4.1.3

	Outputs That Are Not Used	No Outputs
a.	Branch Add	Data Memory
b.	Branch Add, write 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 ALU Mux, 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 to Control to Mux, so the Mux can use the Branch signal to select the PC + 4 input as the new value for PC), and 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.	The two long paths are equal, so both are critical.

4.1.5 One long path is to read the 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, Mux.
b.	The two long paths are equal, so both are critical.

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 4.4.4 and 4.1.5:

a.	The first path (through Regs) is longer.
b.	The two long paths are equal, so both are critical.

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) and 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 (to compare Rs and Rt), and the write port of Registers.
b.	This instruction uses instruction memory, both register read ports, the ALU to add Rd and Rs together, data memory, and the write port in Registers.

4.2.2 New functional blocks needed for this instruction are:

a.	This instruction needs the Zero output of the ALU to be zero-extended to compute the value for Rd. Then we need to add this as another input to the Mux that selects the value to be written into Registers.
b.	None. This instruction can be implemented using existing blocks.

4.2.3 The new control signals are:

a.	We need a new control signal for the Mux that selects between values that can be written into Registers.
b.	None. This instruction can be implemented without adding new control signals. It only requires changes in the Control logic.

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.	1430ps (1130ps + 300ps, ALU is on the critical path)
b.	1130ps. Control latency is now equal to Regs latency, so we have a second critical path (same as the existing one, but going through Control to generate the control signal for the Mux that selects second ALU input). This new critical path has the same latency as the existing one, so the clock cycle is unchanged.

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

	Benefit
a.	We need 5% fewer cycles for a program, but cycle time is 1430 instead of 1130, so we have a speedup of $(1/0.95) \times (1130/1430) = 0.83$, which means we actually have a slowdown.
b.	Speedup is 1 (no change in number of cycles, no change in clock cycle time).

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 $1000 + 200 + 500 + 100 + 2000 + 2 \times 30 + 3 \times 10 = 3890$.

We will compute cost relative to this baseline. The performance relative to this baseline is the speedup 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 + 600 = 4490	4490/3890 = 1.15	1.15/0.83 = 1.39. We are paying significantly more for significantly worse performance, so the cost/performance is a lot worse than with the unmodified processor.
b.	3890 – 400 = 3490	3490/3890 = 0.9	0.9/1 = 0.9. We are reducing cost and getting the same performance, so the cost/performance improves.

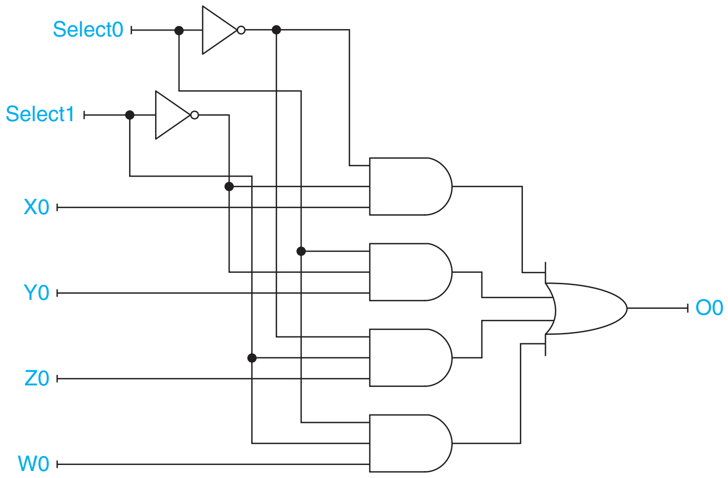
Solution 4.3

4.3.1

a.	Logic only.
b.	Logic only.

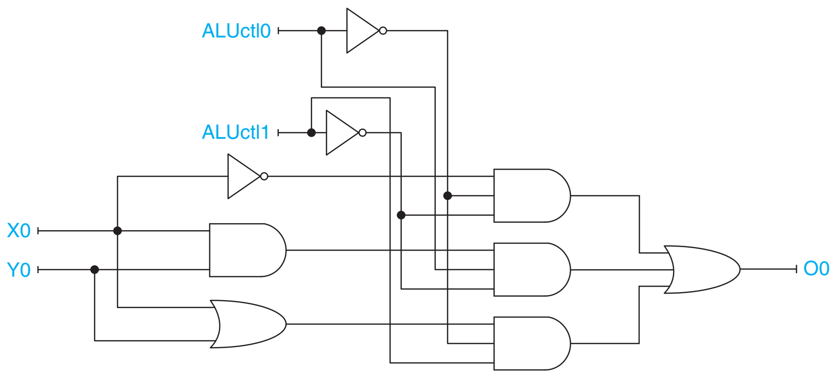
4.3.2

a.



This shows the schematic for the lowermost two bits, where data inputs to the Mux are X (bits X0 through X7), Y, Z, and W. The data output is O (O0-O7). This schematic is repeated 7 more times to handle the remaining 7 data bits.

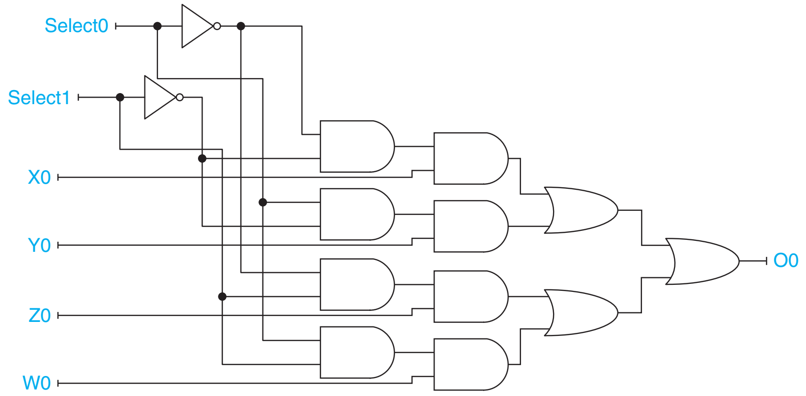
b.



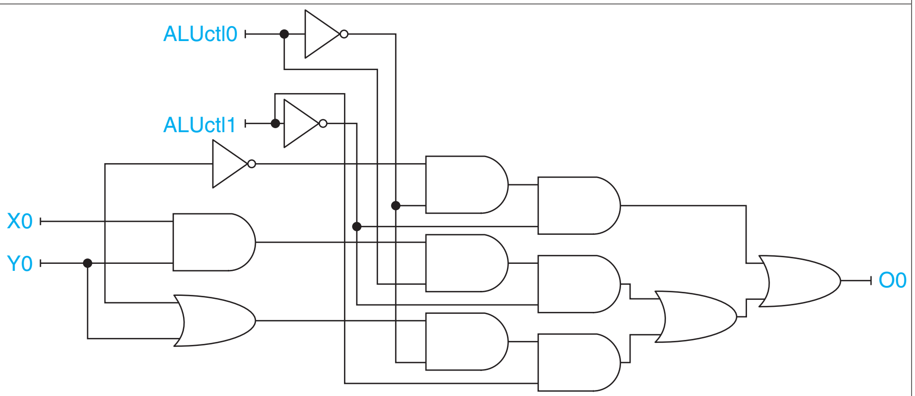
This is the schematic for the lowermost bit, and it needs to be repeated 7 times for the remaining 7 bits. X, Y, and O are the two data inputs and the data output, respectively.

4.3.3

a.



b.



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.

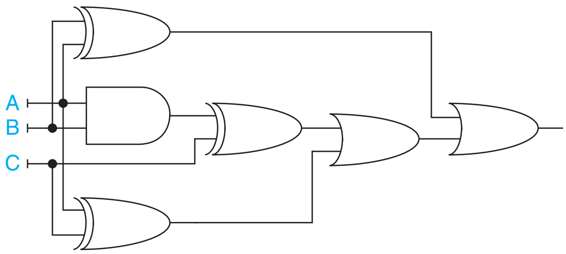
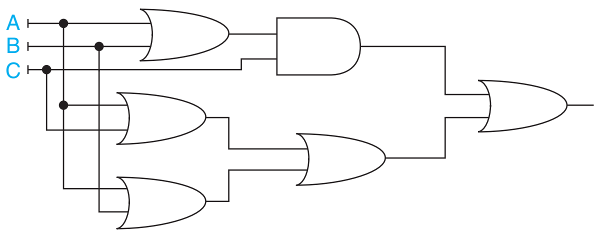
4.3.5 The cost of the implementation is simply the total cost of all its components. Note that there are many correct ways to design the circuit in 4.3.2, and for each solution to 4.3.2 there is a different solution for this problem.

4.3.6

a.	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 (as in 4.3.2) rather than cascades of 2-input gates. The schematic shown for 4.3.2 turns out to already be optimal.
b.	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. We can use De-Morgan's laws to convert sequences of gates into a circuit that only has NOT gates feeding into AND gates which feed into OR gates.

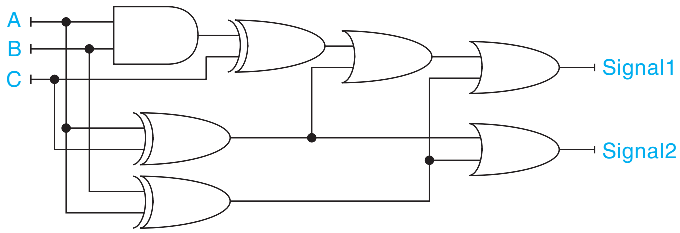
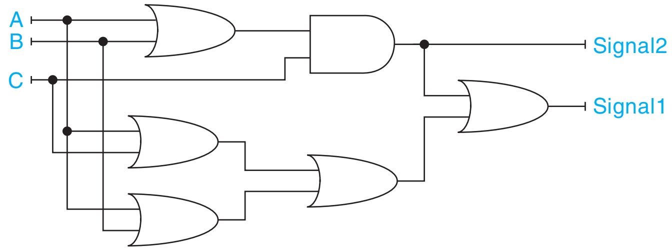
Solution 4.4

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.	 Note: Signal2 is (A AND C) OR (B AND C), which is equal to (A OR B) AND C.

4.4.4

a.	The critical path consists of AND, XOR, OR, and OR, for a total of 82ps.
b.	The critical path consists of three OR gates, for a total of 150ps.

4.4.5

a.	The cost is 1 AND gate, 3 OR gates, and 3 XOR gates, for a total cost of 49.
b.	The cost is 1 AND gate and 5 OR gates, for a total cost of 18.

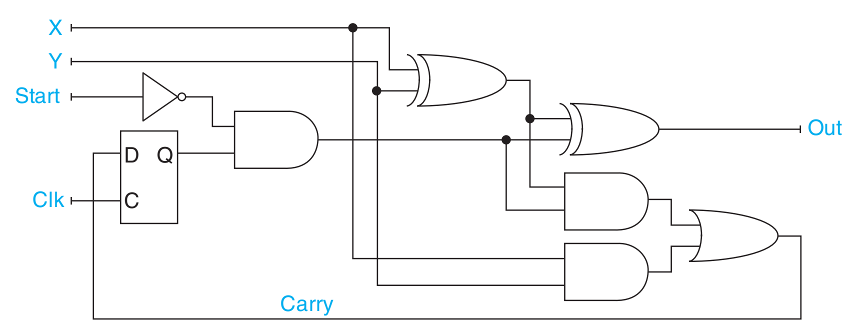
4.4.6 We already computed the cost of the combined circuit. Now we determine the cost of the separate circuits and the savings.

	Combined Cost	Separate Cost	Saved
a.	49	49 (no change)	0%
b.	18	27 (+ 2 AND and 1 OR gate)	(27 – 18)/27 = 33%

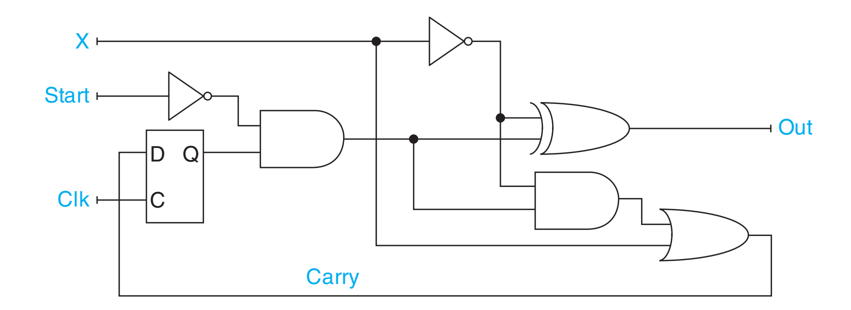
Solution 4.5

4.5.1

a.

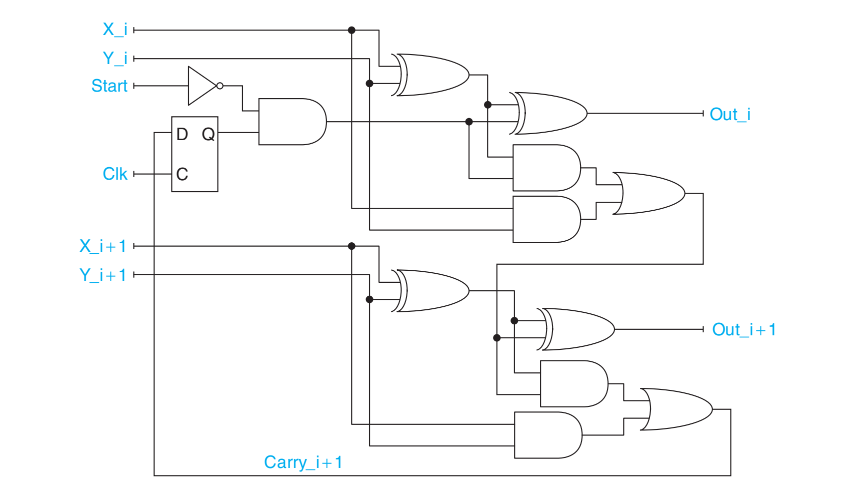


b.

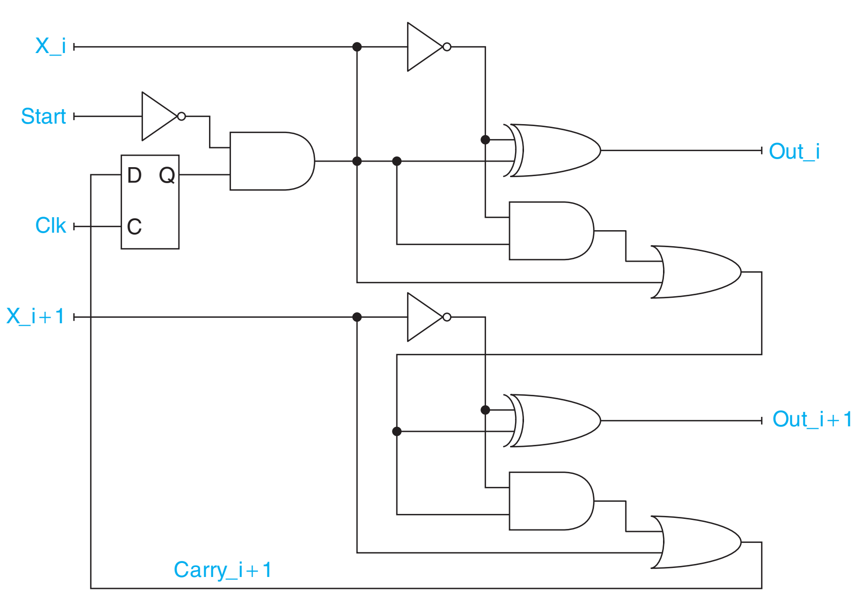


4.5.2

a.



b.



4.5.3

	Cycle Time	Operation Time
a.	76ps (NOT-->AND-->AND-->OR-->D)	32 × 76ps = 2432ps
b.	500ps (NOT-->AND-->AND-->OR-->D)	32 × 500ps = 16000ps

4.5.4

	Cycle Time	Speedup
a.	100ps (NOT-->AND-->AND-->OR-->AND-->OR-->D)	(32 × 76ps)/(16 × 100ps) = 1.52
b.	690ps (NOT-->AND-->AND-->OR-->AND-->OR-->D)	(32 × 500ps)/(16 × 690ps) = 1.45

4.5.5

	Circuit 1	Circuit 2
a.	40 (1 NOT, 3 AND, 1 OR, 2 XOR, 1 D)	64 (1 NOT, 5 AND, 2 OR, 4 XOR, 1 D)
b.	13 (2 NOT, 2 AND, 1 OR, 1 XOR, 1 D)	21 (3 NOT, 3 AND, 2 OR, 2 XOR, 1 D)

4.5.6

	Cost/Performance for Circuit 1	Cost/Performance for Circuit 2	Circuit 1 vs. Circuit 2
a.	$40 \times 32 \times 76 = 97280$	$64 \times 16 \times 100 = 102400$	Cost/performance of Circuit 2 is worse by about 5.3%
b.	$13 \times 32 \times 500 = 208000$	$21 \times 16 \times 690 = 231840$	Cost/performance of Circuit 2 is worse by about 11.5%

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.	200ps
b.	750ps

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 than the latency of the Add unit. We have:

a.	200ps + 15ps + 10ps + 70ps + 20ps = 315ps
b.	750ps + 100ps + 0ps + 200ps + 50ps = 1100ps

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.	200ps + 90ps + 20ps + 90ps + 20ps = 420ps
b.	750ps + 300ps + 50ps + 250ps + 50ps = 1400ps

4.6.4

a.	PC-relative branches.
b.	All instructions except unconditional jumps without a register operand (jal, j).

4.6.5

a.	PC-relative unconditional branch instructions. We saw in 4.6.3 that this is not on the critical path of conditional branches, and it is only needed for PC-relative branches. Note that MIPS does not have actual unconditional branches (BNE zero, zero, Label plays that role so there is no need for unconditional branch opcodes) so for MIPS the answer to this question is actually "None."
b.	All instructions except unconditional jumps without a register operand (jal, j).

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 than or equal to the corresponding path for BNE, so changes in unit latency will not affect this. As a result, we focus on how the unit's latency affects the critical path of BNE:

a.	This unit is not on the critical path, so the only way for this unit to become critical is to increase its latency until the path for address computation through sign extend, shift left, and branch add becomes longer than the path for PCSrc through Registers, Mux, and ALU. The latency of Regs, Mux, and ALU is 200ps and the latency of Sign-extend, Shift-left-2, and Add is 95ps, so the latency of Shift-left-2 must be increased by 105ps or more for it to affect clock cycle time.
b.	This unit is already on the critical path of BNE, so changes in its latency affect the clock cycle time directly. Even if we speed this unit up to have zero latency, the path through Regs, Mux, and ALU will take 300ps and remain a critical path (because Sign-extend, Shift-left-2, and Add also take 300ps).

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.	200ps + 90ps + 20ps + 90ps + 20ps = 420ps
b.	750ps + 300ps + 50ps + 250ps + 50ps = 1400ps

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.	200ps + 90ps + 20ps + 90ps + 250ps + 20ps = 670ps
b.	750ps + 300ps + 50ps + 250ps + 500ps + 50ps = 1900ps

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.	25% + 10% = 35%
b.	30% + 20% = 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.	20% + 25% + 25% + 10% = 80%
b.	10% + 10% + 30% + 20% = 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.	D-Mem has the longest latency, so we reduce its latency from 250ps to 225ps, making the clock cycle 25ps shorter. The speedup achieved by reducing the clock cycle time is then 670ps/645ps = 1.039.
b.	I-Mem has the longest latency, so we reduce its latency from 750ps to 675ps, making the clock cycle 75ps shorter. The speedup achieved by reducing the clock cycle time is then 1900ps/1825ps = 1.041.

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.	If this signal is stuck at zero, an instruction that writes to an odd-numbered register will end up writing to the even-numbered register. So if we place a value of zero in R30 and a value of 1 in R31, and then execute ADD R31, R30, R30 the value of R31 is supposed to be zero. If bit 0 of the Write Register input to the Registers unit is stuck at zero, the value is written to R30 instead and R31 will be 1.
b.	The MIPS architecture requires instructions to be word-aligned (lowermost two bits of the instruction address are always zero). Because of this, we cannot execute an instruction that would set the specified signal to 1, so we cannot test for this stuck-at-0 fault.

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 place a value of zero in R31 and a value of 1 in R30, then use ADD R30, R31, R31 which is supposed to place 0 in R30. If this signal is stuck-at-1, the write goes to R31 instead, so the value in R30 remains 1.
b.	If this signal is stuck-at-1, a branch instruction, such as BNE zero, zero, Label will result in a non-aligned PC (lowermost bit will be 1).

4.8.3

a.	We need to rewrite the program to use only odd-numbered registers.
b.	With this fault, every conditional branch results in a fetch of a misaligned instruction. This prevents any conditional changes in control flow, so the faulty processor is unusable.

4.8.4

a.	To set the MemRead signal to 1 (in order to test for stuck-at-0 fault), we need a load instruction. If MemRead is stuck-at-0, the memory does not get read and the value placed in the register is "random" (whatever happened to be at the output of the memory unit). Unfortunately, this "random" value can be the same as the one already in the register, so this test is not conclusive.
b.	To test for this fault, we need an instruction whose MemRead is 1, so it has to be a load. The instruction also needs to have RegDst set to 0, which is the case for loads. Finally, the instruction needs to have a different result if MemRead is set to 0. For a load, setting MemRead to zero would result in not reading memory at all, so the value placed in the register is "random" (whatever happened to be at the output of the memory unit). Unfortunately, this "random" value can be the same as the one already in the register, so this test is not conclusive.

4.8.5

a.	If Jump is stuck at 0, the PC after a jump is not the jump address. Instead, the PC is either incremented (PC + 4) or computed as if this was a PC-relative branch. To test for this fault, we can place a jump instruction at a low address that jumps to a high address. If the Jump signal is stuck at 0, the PC after the jump will be much lower than it should be. To set the MemRead signal to 1 (in order to test for stuck-at-0 fault), we need a load instruction. If MemRead is stuck-at-0, the memory does not get read and the value placed in the register is "random" (whatever happened to be at the output of the memory unit). Unfortunately, this "random" value can be the same as the one already in the register, so this test is not conclusive.
b.	To test for this fault, we need an instruction whose Jump is 1, so it has to be the jump instruction. However, for the jump instruction the RegDst signal is "don't care" because it does not write to any registers, so the implementation may or may not allow us to set RegDst to 0 so we can test for this fault. As a result, we cannot reliably test for this fault.

4.8.6 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 instructions that trigger the fault.

Solution 4.9

4.9.1

	Binary	Hexadecimal
a.	101011 10000 00100 0000000001100100	AA040064
b.	000000 00010 00011 00001 00000 101010	0043082A

4.9.2

	Read Register 1	Actually Read?	Read Register 2	Actually Read?
a.	16 (10000_b)	Yes	4 (00100_b)	Yes
b.	2 (00010_b)	Yes	3 (00011_b)	Yes

4.9.3

	Read Register 1	Register Actually Written?
a.	Either 4 (00100_b) or 0 (don't know because RegDst is X)	No
b.	1 (00001_b)	Yes

4.9.4

	Control Signal 1	Control Signal 2
a.	ALUSrc = 1	Branch = 0
b.	Jump = 0	RegDst = 1

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.	ALUSrc = I31
b.	Jump = (NOT I31) AND I27

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.	ALUSrc = I31 Branch = I28
b.	RegDst = NOT I31 Jump = RegDst AND I27

Solution 4.10

To solve the problems in this exercise, it helps to first 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 finish generating this signal before the end of the clock cycle. Note that MemWrite is actually a write-enable signal for D-Mem flip-flops, 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.	200ps + 20ps + 90ps + 20ps + 90ps + 250ps + 20ps = 690ps	690ps – 200ps = 490ps
b.	750ps + 50ps + 300ps + 50ps + 250ps + 500ps + 50ps = 1950ps	1950ps – 750ps = 1200ps

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 (30ps > 20ps)	20ps + 90ps + 20ps – 30ps = 100ps
b.	ALUOp (70ps > 50ps)	50ps + 300ps + 50ps – 70ps = 330ps

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.	470ps	470ps	450ps	220ps	470ps	100ps	490ps	110ps	490ps
b.	1150ps	1150ps	1100ps	650ps	1150ps	330ps	1200ps	350ps	1200ps

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 slack for each signal:

	RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
a.	–30ps	–30ps	0ps	20ps	20ps	–100ps	–10ps	10ps	–10ps
b.	50ps	150ps	0ps	–150ps	–50ps	30ps	–100ps	–50ps	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.	ALUOp (–100ps)	690ps	790ps
b.	MemRead (–150ps)	1950ps	2100ps

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.	RegWrite (–10ps) RegDst and Jump (–30ps) ALUOp (–100ps)	170ps at $1/5ps = $34
b.	MemtoReg and ALUSrc (–50ps) MemWrite (–100ps) MemRead (–150ps)	350ps at $1/5ps = $70

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 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.	70ps	70ps	100ps	120ps	120ps	0ps	90ps	110ps	90ps
b.	200ps	300ps	150ps	0ps	100ps	180ps	50ps	100ps	150ps

Solution 4.11

4.11.1

	Sign-Extend	Jump's Shift-Left-2
a.	00000000000000000000000000010100	0001100010000000000001010000
b.	00000000000000000000100000101010	0010000010000010000010101000

4.11.2

	ALUOp[1-0]	Instruction[5-0]
a.	00	010100
b.	10	101010

4.11.3

	New PC	Path
a.	PC + 4	PC to Add (PC + 4) to branch Mux to jump Mux to PC
b.	PC + 4	PC to Add (PC + 4) to branch Mux to jump Mux to PC

4.11.4

	WrReg Mux	ALU Mux	Mem/ALU Mux	Branch Mux	Jump Mux
a.	2 or 0 (RegDst is X)	20	X	PC + 4	PC + 4
b.	1	−128	0	PC + 4	PC + 4

4.11.5

	ALU	Add (PC + 4)	Add (Branch)
a.	−3 and 20	PC and 4	PC + 4 and 20 × 4
b.	−32 and −128	PC and 4	PC + 4 and 2090 × 4

4.11.6

	Read Register 1	Read Register 2	Write Register	Write Data	RegWrite
a.	3	2	X (2 or 0)	X	0
b.	4	2	1	0	1

Solution 4.12

4.12.1

	Pipelined	Single-Cycle
a.	350ps	1250ps
b.	220ps	950ps

4.12.2

	Pipelined	Single-Cycle
a.	1750ps	1250ps
b.	1100ps	950ps

4.12.3

	Stage to Split	New Clock Cycle Time
a.	ID	300ps
b.	EX	210ps

4.12.4

a.	35%
b.	30%

4.12.5

a.	65%
b.	70%

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 an `LW` in

5 cycles, an SW in 4 cycles (no WB), an ALU instruction in 4 cycles (no MEM), and a BEQ in 4 cycles (no WB). So we have the speedup of pipeline:

	Multi-Cycle Execution Time Is X Times Pipelined Execution Time, where X is	Single-Cycle Execution Time Is X Times Pipelined Execution Time, Where X Is
a.	$0.20 \times 5 + 0.80 \times 4 = 4.20$	1250ps/350ps = 3.57
b.	$0.15 \times 5 + 0.85 \times 4 = 4.15$	950ps/220ps = 4.32

Solution 4.13

4.13.1

	Instruction Sequence	Dependences

a.	`I1: SW R16,-100(R6)`
	`I2: LW R4,8(R16)`
	`I3: ADD R5,R4,R4`
		RAW on R4 from I2 to I3

b.	`I1: OR R1,R2,R3`
	`I2: OR R2,R1,R4`
	`I3: OR R1,R1,R2`
		RAW on R1 from I1 to I2 and I3
		RAW on R2 from I2 to I3
		WAR on R2 from I1 to I2
		WAR on R1 from I2 to I3
		WAW on R1 from I1 to I3

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

	Instruction Sequence	

a.	`SW R16,-100(R6)`
	`LW R4,8(R16)`
	`NOP`
	`NOP`
	`ADD R5,R4,R4`
		Delay I3 to avoid RAW hazard on R4 from I2

b.	`OR R1,R2,R3`
	`NOP`
	`NOP`
	`OR R2,R1,R4`
	`NOP`
	`NOP`
	`OR R1,R1,R2`
		Delay I2 to avoid RAW hazard on R1 from I1
		
		Delay I3 to avoid RAW hazard on R2 from I2

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

	Instruction Sequence	

a.	SW R16,-100(R6)
	LW R4,8(R16)
	NOP
	ADD R5,R4,R4
		Delay I3 to avoid RAW hazard on R4 from I2
		Value for R4 is forwarded from I2 now

b.	OR R1,R2,R3 OR R2,R1,R4 OR R1,R1,R2	No RAW hazard on R1 from I1 (forwarded) No RAW hazard on R2 from I2 (forwarded)

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 first 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	Speedup Due to Forwarding
a.	$(7 + 2) \times 250\text{ps} = 2250\text{ps}$	$(7 + 1) \times 300\text{ps} = 2400\text{ps}$	0.94 (This is really a slowdown)
b.	$(7 + 4) \times 180\text{ps} = 1980\text{ps}$	$7 \times 240\text{ps} = 1680\text{ps}$	1.18

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.	SW R16,-100(R6)
	LW R4,8(R16)
	ADD R5,R4,R4
		ALU-ALU forwarding of R4 from I2

b.	OR R1,R2,R3 OR R2,R1,R4 OR R1,R1,R2	ALU-ALU forwarding of R1 from I1 ALU-ALU forwarding of R2 from I2

4.13.6

	No Forwarding	With ALU-ALU Forwarding Only	Speedup with ALU-ALU Forwarding
a.	$(7 + 2) \times 250\text{ps} = 2250\text{ps}$	$7 \times 290\text{ps} = 2030\text{ps}$	1.11
b.	$(7 + 4) \times 180\text{ps} = 1980\text{ps}$	$7 \times 210\text{ps} = 1470\text{ps}$	1.35

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.	SW R16,12(R6)
	LW R16,8(R6)
	BEQ R5,R4,Lbl
	ADD R5,R1,R4
	SLT R5,R15,R4
		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
			11

b.	SW R2,0(R3)
	OR R1,R2,R3
	BEQ R2,R0,Lbl
	ADD R1,R4,R3
		IF ID EX MEM WB
		IF ED EX MEM WB
		IF ID EX MEM WB
		*** IF ID EX MEM WB
			9

We cannot 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 finishes one cycle earlier (one less stage to go through). If there were data hazards from loads to other instructions, the change would help eliminate some stall cycles.

	Instructions Executed	Cycles with 5 Stages	Cycles with 4 Stages	Speedup
a.	5	4 + 5 = 9	3 + 5 = 8	9/8 = 1.13
b.	4	4 + 4 = 8	3 + 4 = 7	8/7 = 1.14

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	Speedup
a.	5	1	$4 + 5 + 1 \times 2 = 11$	$4 + 5 + 1 \times 1 = 10$	11/10 = 1.10
b.	4	1	$4 + 4 + 1 \times 2 = 10$	$4 + 4 + 1 \times 1 = 9$	10/9 = 1.11

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	Speedup
a.	200ps (IF)	210ps (MEM + 20ps)	$(9 \times 200)/(8 \times 210) = 1.07$
b.	200ps (ID, EX, MEM)	220ps (MEM + 20ps)	$(8 \times 200)/(7 \times 220) = 1.04$

4.14.5

	New ID Latency	New EX Latency	New Cycle Time	Old Cycle Time	Speedup
a.	180ps	140ps	200ps (IF)	200ps (IF)	$(11 \times 200)/(10 \times 200) = 1.10$
b.	300ps	190ps	300ps (ID)	200ps (ID, EX, MEM)	$(10 \times 200)/(9 \times 300) = 0.74$

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 speedup 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)	Speedup
a.	$4 + 5 + 1 \times 2 = 11$	$11 \times 200ps = 2200ps$	$4 + 5 + 1 \times 3 = 12$	$12 \times 200ps = 2400ps$	0.92
b.	$4 + 4 + 1 \times 2 = 10$	$10 \times 200ps = 2000ps$	$4 + 4 + 1 \times 3 = 11$	$11 \times 200ps = 2200ps$	0.91

Solution 4.15

4.15.1

a.	This instruction behaves like a normal load until the end of the MEM stage. After that, it behaves like an ADD, so we need another stage after MEM to compute the result, and we need additional wiring to get the value of Rt to this stage.
b.	This instruction behaves like a load until the end of the MEM stage. After that, we need another stage to compare the value against Rt. We also need to add an input to the PC Mux that takes the value of Rd, and the Mux select signal must now include the result of the new comparison. We also need an extra read port in Registers because the instruction needs three registers to be read.

4.15.2

a.	We need to add a control signal that selects what the new stage does (just pass the value from memory through, or add the register value to it).
b.	We need a control signal similar to the existing "Branch" signal to control whether or not the new comparison is allowed to affect the PC. We also need to add one bit to the control signal that selects whether the target address is PC + 4 + Offs or the register value.

4.15.3

a.	The addition of a new stage 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. Additionally, this instruction produces its result only at the end of the new stage, so even with forwarding it introduces a data hazard that requires a two-cycle stall if the ADDM instruction is immediately followed by a data-dependent instruction.
b.	The addition of a new stage 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. The instruction itself creates a control hazard that leaves the next PC unknown until the BEQM instruction leaves the new stage, which is two cycles longer than for a normal BEQ.

4.15.4

a.	`LW  Rd,Offs(Rs)` `ADD Rd,Rt,Rd`	E.g., ADDM can be used when trying to compute a sum of array elements.

b.	`LW  Rtmp,Offs(Rs)`
	`BNE Rtmp,Rt,Skip`
	`JR  Rd`
	`Skip:`
		E.g., BEQM can be used when trying to determine if an array has an element with a specific value.

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 instructions 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	Speedup
a.	30 – (2 – 1) + 2 = 31	30	0.97
b.	40 – (3 – 1) + 1 = 39	40	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 fields of the instruction word (even for instructions whose format does not use these fields). The EX/MEM register keeps control signals for the 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 field), 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.	R6, R16	R6, R16
b.	R1, R0	R1, R0

4.16.3

	EX	MEM
a.	−100 + R6	Write value to memory
b.	R1 OR R0	Nothing

4.16.4

	Loop							
a.	2: LW R2,16(R2)	WB						
	2: SLT R1,R2,R4	EX	MEM	WB				
	2: BEQ R1,R9,Loop	ID	EX	MEM	WB			
	3: ADD R1,R2,R1	IF	ID	EX	MEM	WB		
	3: LW R2,0(R1)		IF	ID	EX	MEM	WB	
	3: LW R2,16(R2)			IF	ID	***	EX	MEM
	3: SLT R1,R2,R4				IF	***	ID	***
	3: BEQ R1,R9,Loop						IF	***

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

4.16.5 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 cycle, 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.	7	1	14%
b.	8	2	0%

4.16.6 The address of that first 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.	18% (60% of 30%)
b.	6% (60% of 10%)

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.	40%
b.	60%

4.17.3 Of these instructions, only LW and SW use the data memory. We have:

a.	30% (25% + 5%)
b.	30% (20% + 10%)

4.17.4 The clock cycle time of a single-cycle is the sum of all latencies for the logic of all five stages. The clock cycle time of a pipelined datapath is the maximum latency of the five 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	Speedup
a.	760ps	215ps	3.53
b.	850ps	215ps	3.95

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.	610ps	215ps
b.	650ps	215ps

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 speedup is below 1 (there is a slowdown):

	Instructions with 5-Stage	Instructions with 4-Stage	Speedup
a.	$1.00 \times I$	$1.00 \times I + 0.5 \times (0.25 + 0.05) \times I = 1.150 \times I$	0.87
b.	$1.00 \times I$	$1.00 \times I + 0.5 \times (0.20 + 0.10) \times I = 1.150 \times I$	0.87

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 = 1, ALUOp = 00, RegDst = 0	Branch = 0, MemWrite = 0, MemRead = 1	MemtoReg = 0, 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 MEM, used in MEM
b.	Generated in ID, used in MEM	Generated in ID, used in WB

4.18.5

a.	None. PCSRc is only 1 for a taken branch, and ALUsrc is 0 for PC-relative branches.
b.	None. Branch is only 1 for conditional branches, and conditional branches do not write registers.

4.18.6 Signal 2 goes back through 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 the 1st and 2nd next instruction. Dependences to only the 2nd next instruction result in one stall cycle. We have:

	CPI	Stall Cycles
a.	$1 + 0.35 \times 2 + 0.15 \times 1 = 1.85$	46% (0.85/1.85)
b.	$1 + 0.35 \times 2 + 0.25 \times 1 = 1.95$	49% (0.95/1.95)

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 these dependences cause only one stall cycle, so we have:

	CPI	Stall Cycles
a.	$1 + 0.20 = 1.20$	17% (0.20/1.20)
b.	$1 + 0.10 = 1.1$	13% (0.15/1.15)

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

	EX/MEM	MEM/WB	Fewer Stall Cycles with
a.	0.2 + 0.05 + 0.1 + 0.1 = 0.45	0.05 + 0.2 + 0.1 = 0.35	MEM/WB
b.	0.1 + 0.15 + 0.1 + 0.05 = 0.4	0.2 + 0.1 + 0.05 = 0.35	MEM/WB

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	Speedup
a.	1.85 × 150ps = 277.5ps	1.20 × 150ps = 180ps	1.54
b.	1.95 × 300ps = 585ps	1.1 × 350ps = 385ps	1.52

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 speedup over full forwarding:

	With Full Forwarding	Time-Travel Forwarding	Speedup
a.	1.20 × 150ps = 180ps	1 × 250ps = 250ps	0.72
b.	1.1 × 350ps = 385ps	1 × 450ps = 450ps	0.86

4.19.6

	EX/MEM	MEM/WB	Shorter Time per Instruction with
a.	1.45 × 150ps = 217.5	1.35 × 150ps = 202.5ps	MEM/WB
b.	1.4 × 330ps = 462	1.35 × 320ps = 432ps	MEM/WB

Solution 4.20

4.20.1

	Instruction Sequence	RAW	WAR	WAW

a.	`I1: ADD R1,R2,R1`
	`I2: LW R2,0(R1)`
	`I3: LW R1,4(R1)`
	`I4: OR R3,R1,R2`
		(R1) I1 to I2, I3
		(R2) I2 to I4
		(R1) I3 to I4
			(R2) I1 to I2
			(R1) I1, I2 to I3
				(R1) I1 to I3

b.	I1: LW R1,0(R1) I2: AND R1,R1,R2 I3: LW R2,0(R1) I4: LW R1,0(R3)	(R1) I1 to I2 (R1) I2 to I3	(R1) I1 to I2 (R2) I2 to I3 (R1) I3 to I4	(R1) I1 to I2 (R1) I2 to I4

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 3 instructions becomes a hazard:

	Instruction Sequence	With Forwarding	Without Forwarding
a.	I1: ADD R1,R2,R1 I2: LW R2,0(R1) I3: LW R1,4(R1) I4: OR R3,R1,R2	(R1) I3 to I4	(R1) I1 to I2, I3 (R2) I2 to I4 (R1) I3 to I4
b.	I1: LW R1,0(R1) I2: AND R1,R1,R2 I3: LW R2,0(R1) I4: LW R1,0(R3)	(R1) I1 to I2	(R1) I1 to I2 (R1) I2 to I3

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: ADD R1,R2,R1 I2: LW R2,0(R1) I3: LW R1,4(R1) I4: OR R3,R1,R2	(R2) I2 to I4 (R1) I3 to I4	(R1) I1 to I2, I3 (R2) I2 to I4 (R1) I3 to I4
b.	I1: LW R1,0(R1) I2: AND R1,R1,R2 I3: LW R2,0(R1) I4: LW R1,0(R3)	(R1) I1 to I2	(R1) I1 to I2 (R1) I2 to I3

4.20.4

	Instruction Sequence	RAW

a.	`I1: ADD R1,R2,R1`
	`I2: LW  R2,0(R1)`
	`I3: LW  R1,4(R1)`
	`I4: OR  R3,R1,R2`
		(R1) I1 to I2 (30 overrides –1)

b.	`I1: LW  R1,0(R1)`
	`I2: AND R1,R1,R2`
	`I3: LW  R2,0(R1)`
	`I4: LW  R1,0(R3)`
		(R1) I1 to I2 (0 overrides 4)

4.20.5 A register modification 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: ADD R1,R2,R1`
	`I2: LW  R2,0(R1)`
	`I3: LW  R1,4(R1)`
	`    Stall`
	`I4: OR  R3,R1,R2`
		R1 = 30 (Stall and after)
		R2 = 0 (I4 and after)
		R1 = 0 (after I4)
		
		R3 = 30 (after I4)
			R0 = 0
			R1 = 0
			R2 = 0
			R3 = 30

b.	`I1: LW  R1,0(R1)`
	`    Stall`
	`I2: AND R1,R1,R2`
	`I3: LW  R2,0(R1)`
	`I4: LW  R1,0(R3)`
		R1 = 0 (I3 and after)
		
		R1 = 4 (after I4)
		R2 = 0
		R1 = 0
			R0 = 0
			R1 = 0
			R2 = 0
			R3 = 3000

4.20.6

	Instruction Sequence with Forwarding Stalls	Correct Execution	Sequence with NOPs

a.	`I1: ADD R1,R2,R1`
	`I2: LW  R2,0(R1)`
	`I3: LW  R1,4(R1)`
	`    Stall`
	`I4: OR  R3,R1,R2`
		`I1: ADD R1,R2,R1`
		`Stall`
		`Stall`
		`I2: LW  R2,0(R1)`
		`I3: LW  R1,4(R1)`
		`    Stall`
		`    Stall`
		`I4: OR  R3,R1,R2`
			`ADD R1,R2,R1`
			`NOP`
			`NOP`
			`LW  R2,0(R1)`
			`LW  R1,4(R1)`
			`NOP`
			`NOP`
			`OR  R3,R1,R2`

b.	I1: LW R1,0(R1)
	Stall
	I2: AND R1,R1,R2
	I3: LW R2,0(R1)
	I4: LW R1,0(R3)
		I1: LW R1,0(R1)
		Stall
		Stall
		I2: AND R1,R1,R2
		Stall
		Stall
		I3: LW R2,0(R1)
		I4: LW R1,0(R3)
			LW R1,0(R1)
			NOP
			NOP
			AND R1,R1,R2
			NOP
			NOP
			LW R2,0(R1)
			LW R1,0(R3)

Solution 4.21

4.21.1

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

b.	LW R2,0(R1)
	NOP
	NOP
	AND R1,R2,R1
	LW R3,0(R2)
	NOP
	LW R1,0(R1)
	NOP
	NOP
	SW R1,0(R2)

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 fill some NOP slots with such instructions. We can also use R7 to eliminate WAW or WAR dependences so we can have more instructions to move up.

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

b.	`I1: LW  R2,0(R1)`
	`NOP`
	`NOP`
	`I2: AND R1,R2,R1`
	`I3: LW  R3,0(R2)`
	`NOP`
	`I4: LW  R1,0(R1)`
	`NOP`
	`NOP`
	`I5: SW  R1,0(R2)`
		No improvement is possible. There is a chain of RAW dependences from I1 to I2 to I4 to I5, and each step in the chain has to be separated by two instructions.

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.	Code executes correctly (for both loads, there is no RAW dependence between the load and the next instruction).
b.	I1 gets the value of R2 from before I1, not from I1 as it should. Also, I5 gets the value of R1 from I1, not from I4 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 are ALUin1 and ALUin2, which control Muxes which select the first and second input of the ALU. The three possible values for ALUin1 or ALUin2 are 0 (no forwarding), 1 (forward ALU output from previous instruction), or 2 (forward data value for second-previous instruction). We have:

	Instruction Sequence	First Five Cycles 1 2 3 4 5	Signals

a.	ADD R5,R2,R1
	LW R3,4(R5)
	LW R2,0(R2)
	OR R3,R5,R3
	SW R3,0(R5)
		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 = 1, ALUin2 = 0
			5: PCWrite = 1, ALUin1 = 0, ALUin2 = 0

b.	LW R2,0(R1)
	AND R1,R2,R1
	LW R3,0(R2)
	LW R1,0(R1)
	SW R1,0(R2)
		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 = 2, 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 register number (this is the bottommost output of the EX/MEM pipeline register). The additional inputs to the hazard detection unit are register Rd from the ID/EX pipeline register and the output number of the output register from the EX/MEM pipeline register. The Rt field from the ID/EX register is already an input of the hazard detection unit in Figure 4.60.

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

4.21.6 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 Five Cycles 1 2 3 4 5	Signals

a.	ADD R5,R2,R1
	LW R3,4(R5)
	LW R2,0(R2)
	OR R3,R5,R3
	SW R3,0(R5)
		IF ID EX MEM WB
		IF ID *** ***
		IF *** ***
		***
			1: PCWrite = 1
			2: PCWrite = 1
			3: PCWrite = 1
			4: PCWrite = 0
			5: PCWrite = 0

b.	LW R2,0(R1)
	AND R1,R2,R1
	LW R3,0(R2)
	LW R1,0(R1)
	SW R1,0(R2)
		IF ID EX MEM WB
		IF ID *** ***
		IF *** ***
		***
			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 R2,0(R2)	IF	ID	EX	MEM	WB									
	BEQ R2,R0,Label (T)		IF	ID	***	EX	MEM	WB							
	LW R2,0(R2)			IF	***	ID	EX	MEB	WB						
	BEQ R2,R0,Label (NT)					IF	ID	***	EX	MEM	WB				
	OR R2,R2,R3									IF	ID	EX	MEB	WB	
	SW R2,0(R5)										IF	ID	EX	MEB	WB
b.	LW R2,0(R1)	IF	ID	EX	MEM	WB									
	BEQ R2,R0,Label2 (NT)		IF	ID	***	EX	MEB	WB							
	LW R3,0(R2)						IF	ID	EX	MEB	WB				
	BEQ R3,R0,Label1 (T)							IF	ID	***	EX	MEB	WB		
	BEQ R2,R0,Label2 (T)								IF	***	ID	EX	MEB	WB	
	SW R1,0(R2)										IF	ID	EX	MEB	WB

4.22.2

	Executed Instructions	Pipeline Cycles 1	2	3	4	5	6	7	8	9	10	11	12	13	14
a.	LW R2,0(R2)	IF	ID	EX	MEM	WB									
	BEQ R2,R0,Label (T)		IF	ID	***	EX	MEB	WB							
	OR R2,R2,R3			IF	***	ID	EX	MEB	WB						
	LW R2,0(R2)					IF	ID	***	EX	MEM	WB				
	BEQ R2,R0,Label (NT)						IF	***	ID	EX	MEM	WB			
	OR R2,R2,R3								IF	ID	EX	MEM	WB		
	SW R2,0(R5)										IF	ID	EX	MEB	WB
b.	LW R2,0(R1)	IF	ID	EX	MEM	WB									
	BEQ R2,R0,Label2 (NT)		IF	ID	***	EX	MEM	WB							
	LW R3,0(R2)			IF	***	ID	EX	MEB	WB						
	BEQ R3,R0,Label1 (T)						IF	ID	EX	MEM	WB				
	ADD R1,R3,R1							IF	ID	EX	MEM	WB			
	BEQ R2,R0,Label2 (T)								IF	ID	EX	MEM	WB		
	LW R3,0(R2)									IF	ID	EX	MEM	WB	
	SW R1,0(R2)										IF	ID	EX	MEM	WB

4.22.3

a.	`Label:  LW  R2,0(R2)`
	`        BEZ R2,Label ; Taken once, then not taken`
	`        OR  R2,R2,R3`
	`        SW  R2,0(R5)`

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

4.22.4 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 shown 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:

		Pipeline Cycles														
	Executed Instructions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
a.	`LW R2,0(R2)`	IF	ID	EX	MEM	WB										
	`BEQ R2,R0,Label (T)`		IF	***	***	ID	EX	MEB	WB							
	`LW R2,0(R2)`					IF	ID	EX	MEB	WB						
	`BEQ R2,R0,Label (NT)`						IF	***	***	ID	EX	MEB	WB			
	`OR R2,R2,R3`									IF	ID	EX	MEB	WB		
	`SW R2,0(R5)`										IF	ID	EX	MEB	WB	
b.	`LW R2,0(R1)`	IF	ID	EX	MEM	WB										
	`BEQ R2,R0,Label2 (NT)`		IF	***	***	ID	EX	MEM	WB							
	`LW R3,0(R2)`						IF	ID	EX	MEB	WB					
	`BEQ R3,R0,Label1 (T)`							IF	***	***	ID	EX	MEB	WB		
	`BEQ R2,R0,Label2 (T)`										IF	ID	EX	MEM	WB	
	`SW R1,0(R2)`											IF	ID	EX	MEB	WB

Now the speedup can be computed as:

a.	14/14 = 1
b.	14/15 = 0.93

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 − 0.45) × 0.25 = 0.41
b.	3 × (1 − 0.65) × 0.08 = 0.08

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 − 0.55) × 0.25 = 0.34
b.	3 × (1 − 0.35) × 0.08 = 0.16

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 − 0.85) × 0.25 = 0.113
b.	3 × (− 0.98) × 0.08 = 0.005

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	Speedup from Conversion
a.	1 + 3 × (1 − 0.85) × 0.25 = 1.113	1 + 3 × (1 − 0.85) × 0.25 × 0.5 = 1.056	1.113/1.056 = 1.054
b.	1 + 3 × (1 − 0.98) × 0.08 = 1.005	1 + 3 × (1 − 0.98) × 0.08 × 0.5 = 1.002	1.005/1.002 = 1.003

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	Speedup from Conversion
a.	1.113	$1 + (1 + 3 \times (1 - 0.85)) \times 0.25 \times 0.5 = 1.181$	$1.113/1.181 = 0.94$
b.	1.015	$1 + (1 + 3 \times (1 - 0.98)) \times 0.08 \times 0.5 = 1.042$	$1.005/1.042 = 0.96$

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.	$B \times 0.85$	$B \times 0.05$	$(B \times 0.05)/(B \times 0.20) = 0.25$ (25%)
b.	$B \times 0.98$	$B \times 0.18$	$(B \times 0.18)/(B \times 0.20) = 0.90$ (90%)

Solution 4.24

4.24.1

	Always Taken	Always Not-taken
a.	2/4 = 50%	2/4 = 50%
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, NT	0,1,2,1	I,I,I,C	25%
b.	T, NT, T, T	0,1,0,1	I,C,I,I	25%

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

	Outcomes	Predictor Value at Time of Prediction	Correct or Incorrect (in Steady State)	Accuracy in Steady State
a.	T,T,NT,NT	1st occurrence: 0,1,2,1 2nd occurrence: 0,1,2,1	I,I,I,C	25%
b.	T, NT, T, T, NT	1st occurrence: 0,1,0,1,2 2nd occurrence: 1,2,1,2,3 3rd occurrence: 2,3,2,3,3 4th occurrence: 2,3,2,3,3	C,I,C,C,I	60%

4.24.4 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 first prediction will be incorrect, so the predictor's state is inverted and after that the predictions are always correct. Overall, there is no warm-up period for the given pattern, and the warm-up period for the opposite pattern is only one branch.

Solution 4.25

4.25.1

	Instruction 1	Instruction 2
a.	Invalid target address (EX)	Invalid data address (MEM)
b.	Invalid target address (EX)	Invalid data address (MEM)

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 first instruction must be converted to `NOPs`. As a result, the second instruction never completes and does not affect pipeline state. In the cycle that immediately follows the cycle in which the exception is detected, the processor will fetch the first instruction of the exception handler.

4.25.4

	Handler Address
a.	0×1000E230
b.	0×678A0000

The first 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 an `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 computes 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 first save the general-purpose register (so we can restore it later), load the Cause register into it, add the address of the vector table to it, use the result as an address for a load that gets the address of the right exception handler from memory, and finally jump to that handler.

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.	ID	IF.Flush = ID.Flush = 1, PCSel = Exc
b.	EX	IF.Flush = ID.Flush = EX.Flush = 1, PCSel = Exc

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 overflow is detected in EX, each exception results in a 3-cycle delay (IF, ID, and EX are cancelled). By moving overflow into MEM, we add one more cycle to this delay. To compute the speedup, 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	Speedup
a.	250ps	220ps	250ps × 100,003	220ps × 100,004	1.13635
b.	200ps	175ps	200ps × 100,003	175ps × 100,004	1.14285

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 overflow 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 first instruction of the exception handler, in ID we have an `NOP` (instead of I4, marked), in EX we have an `NOP` (instead of I3), and I1 and I2 still continue through the pipeline normally:

	Branch and Delay Slot	Pipeline

a.	`I1: BEQ R5,R4,Label`
	`I2: SLT R5,R15,R4`
	`I3: Invalid`
	`I4: Something`
	`I5: Handler`
		`IF ID EX MEM WB`
		`   IF ID EX  MEM`
		`      IF ID  *EX`
		`         IF  *ID`
		`             IF`

b.	I1: BEQ R1,R0,Label I2: LW R1,0(R1) 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 first instruction of the exception handler for the exception triggered by I2.

	Branch and Delay Slot	Branch and Delay Slot
a.	I1: BEQ R5,R4,Label I2: SLT R5,R15,R4 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 R1,R0,Label I2: LW R1,0(R1) 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 is 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 overflow 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 cannot 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  R2,R0,R0
Again: BEQ  R2,R8,End
       ADD  R3,R2,R9
       LW   R4,0(R3)
       SW   R4,1(R3)
       ADDI R2,R2,2
       BEQ  R0,R0,Again
End:
```

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

4.28.2

a.

Instructions	Pipeline
ADD R2,R0,R0	IF ID EX ME WB
BEQ R2,R8,End	IF ID ** EX ME WB
ADD R3,R2,R9	IF ** ID EX ME WB
LW R4,0(R3)	IF ** ID ** EX ME WB
SW R4,1(R3)	IF ** ID EX ME WB
ADDI R2,R2,2	IF ** ID EX ME WB
BEQ R0,R0,Again	IF ID EX ME WB
BEQ R2,R8,End	IF ID ** EX ME WB
ADD R3,R2,R9	IF ** ID EX ME WB
LW R4,0(R3)	IF ** ID ** EX ME WB
SW R4,1(R3)	IF ** ID EX ME WB
ADDI R2,R2,2	IF ** ID EX ME WB
BEQ R0,R0,Again	IF ID EX ME WB
BEQ R2,R8,End	IF ID ** EX ME WB

```
ADD  R2,R0,R0      IF ID EX ME WB
BEQ  R2,R8,End     IF ID ** EX ME WB
ADD  R3,R2,R9         IF ** ID EX ME WB
LW   R4,0(R3)         IF ** ID ** EX ME WB
SW   R4,1(R3)               IF ** ID EX ME WB
ADDI R2,R2,2                IF ** ID EX ME WB
BEQ  R0,R0,Again                  IF ID EX ME WB
BEQ  R2,R8,End                    IF ID ** EX ME WB
ADD  R3,R2,R9                        IF ** ID EX ME WB
LW   R4,0(R3)                        IF ** ID ** EX ME WB
SW   R4,1(R3)                              IF ** ID EX ME WB
ADDI R2,R2,2                               IF ** ID EX ME WB
BEQ  R0,R0,Again                                 IF ID EX ME WB
BEQ  R2,R8,End                                   IF ID ** EX ME WB
```

b.

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

4.28.3 The only way to execute two 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  R2,R0,R0`
	`Again:  ADD  R3,R2,R9`
	`        BEQ  R2,R8,End`
	`        LW   R4,0(R3)`
	`        ADDI R2,R2,2`
	`        SW   R4,1(R3)`
	`        BEQ  R0,R0,Again`
	`End:`
		Note that we are now computing a + i before we check whether we should continue the loop. This is OK because we are allowed to “trash” R3. If we exit the loop one extra instruction is executed, but if we stay in the loop we allow both of the memory instructions to execute in parallel with other instructions.

b.	`        ADD  R5,R0,R0`
	`Again:  ADD  R10,R5,R1`
	`        BEQ  R5,R6,End`
	`        LW   R11,0(R10)`
	`        ADD  R12,R5,R2`
	`        LW   R10,1(R10)`
	`        ADDI R5,R5,2`
	`        SUB  R10,R11,R10`
	`        SW   R10,0(R12)`
	`        BEQ  R0,R0,Again`
	`End:`
		Note that we are now computing a + i before we check whether we should continue the loop. This is OK because we are allowed to “trash” R10. If we exit the loop one extra instruction is executed, but if we stay in the loop we allow both of the memory instructions to execute in parallel with other instructions.

4.28.4

Instructions | Pipeline

a.

```
ADD  R2,R0,R0       IF ID EX ME WB
ADD  R3,R2,R9       IF ID ** EX ME WB
BEQ  R2,R8,End         IF ** ID EX ME WB
LW   R4,0(R3)          IF ** ID EX ME WB
ADDI R2,R2,2                 IF ID EX ME WB
SW   R4,1(R3)                IF ID EX ME WB
BEQ  R0,R0,Again                IF ID EX ME WB
ADD  R3,R2,R9                   IF ID ** EX ME WB
BEQ  R2,R8,End                     IF ** ID EX ME WB
LW   R4,0(R3)                      IF ** ID EX ME WB
ADDI R2,R2,2                             IF ID EX ME WB
SW   R4,1(R3)                            IF ID EX ME WB
BEQ  R0,R0,Again                            IF ID EX ME WB
ADD  R3,R2,R9                               IF ID ** EX ME WB
BEQ  R2,R8,End                                 IF ** ID EX ME WB
```

b.

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

4.28.5

	CPI for 1-Issue	CPI for 2-Issue	Speedup
a.	1 (no data hazards)	0.83 (5 cycles for 6 instructions). In every iteration SW can execute in parallel with the next instruction.	1.20
b.	1.11 (10 cycles per 9 instructions). There is 1 stall cycle in each iteration due to a data hazard between the second LW and the next instruction (SUB).	1.06 (19 cycles per 18 instructions). Neither of the two LW instructions can execute in parallel with another instruction, and SUB stalls because it depends on the second LW. The SW instruction executes in parallel with ADDI in even-numbered iterations.	1.05

4.28.6

	CPI for 1-Issue	CPI for 2-Issue	Speedup
a.	1	0.67 (4 cycles for 6 instructions). In every iteration ADD and LW cannot execute in the same cycle because of a data dependence. The rest of the instructions can execute in pairs.	1.49
b.	1.11	0.83 (15 cycles per 18 instructions). In all iterations, SUB is stalled because it depends on the second LW. The only instructions that execute in odd-numbered iterations as a pair are ADDI and BEQ. In even-numbered iterations, only the two LW instructions cannot execute as a pair.	1.34

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 forwarded from prior instructions. We assume that register writes happen in the first half of the cycle and the register reads happen in the second half.

	Loop	Pipeline Stages	Useful Reads	% Useful

a.	ADD R2,R2,R3
	BEQ R2,zero,Loop
	ADDI R1,R1,4
	LW R2,0(R1)
	LW R3,16(R1)
	ADD R2,R2,R1
	ADD R2,R2,R3
	BEQ R2,zero,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 (R1 fw)
			0 (R1 fw)
			1 (R1,R2 fw)
			0 (R2,R3 fw)
			1 (R2 fw)
				15%
				
				(3/(5×4))

b.	AND R1,R1,R2
	LW R2,0(R2)
	BEQ R1,zero,Loop
	LW R1,0(R1)
	AND R1,R1,R2
	LW R2,0(R2)
	BEQ R1,zero,Loop
		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
			
			0 (R1 fw)
			0 (R1,R2 fw)
			1
			1 (R1 fw)
				12.5%
				
				(2/(4×4))

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

	Loop	Pipeline Stages	Useful Reads	% Useful

a.	ADD R2,R2,R3
	BEQ R2,zero,Loop
	ADDI R1,R1,4
	LW R2,0(R1)
	LW R3,16(R1)
	ADD R2,R2,R1
	ADD R2,R2,R3
	BEQ R2,zero,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 (R1 fw)
			0 (R1 fw)
			0 (R1,R2 fw)
			0 (R2,R3 fw)
			1 (R2 fw)
				
				5.5%
				
				(2/(6 × 6))

b.	AND R1,R1,R2
	LW R2,0(R2)
	BEQ R1,zero,Loop
	LW R1,0(R1)
	AND R1,R1,R2
	LW R2,0(R2)
	BEQ R1,zero,Loop
	LW R1,0(R1)
	AND R1,R1,R2
	LW R2,0(R2)
	BEQ R1,zero,Loop
	LW R1,0(R1)
	AND R1,R1,R2
	LW R2,0(R2)
	BEQ R1,zero,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
			
			0 (R1 fw)
			0 (R1,R2 fw)
			0 (R2 fw)
			1 (R1 fw)
			0 (R1 fw)
			0 (R1,R2 fw)
			1
			1 (R1 fw)
			0 (R1 fw)
			0 (R1,R2 fw)
			0 (R2 fw)
			1 (R1 fw)
				
				6.7%
				
				(4/(10 × 6))

4.29.3

	2 Ports Used	3 Ports Used
a.	1 cycle out of 6 (16.7%)	Never (0%)
b.	3 cycles out of 10 (30%)	1 cycle out of 10 (10%)

4.29.4

	Unrolled and Scheduled Loop	Comment

a.	NOP
	Loop: LW R2,8(R1)
	LW R3,24(R1)
	ADDI R1,R1,8
	ADD R2,R2,R1
	ADD R2,R2,R3
	BEQ R2,zero,Loop
		We are able to complete one iteration of the unrolled loop every 4 cycles. Both loads and adds that come from the first original iteration of the unrolled loop can be eliminated (they are only used to compute R2 for BEQ, which is removed). We combine both ADDI instructions and then schedule the unrolled loop to execute in four cycles per (unrolled) iteration, which is optimal. Note the NOP before the loop, which is needed to ensure that BEQ always executes together with the first LW of the next iteration.

b.	<pre> NOP
	Loop: LW R1,0(R1)
	        LW R10,0(R2)
	        NOP
	        AND R1,R1,R2
	        LW R1,0(R1)
	        AND R1,R1,R10
	        LW R2,0(R10)
	        BEQ R1,zero,Loop</pre>
		We are able to execute one iteration of the unrolled loop in 6 cycles, which is optimal. Note the NOP before the loop, which is needed to ensure that BEQ always executes together with the first LW of the next iteration.

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 speedup computation because the two versions of the loop do not execute the same instructions.

	Original Loop	Unrolled Loop	Speedup
a.	$5 \times 2 = 10$	4	2.5
b.	$4 \times 2 = 8$	6	1.3

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 an `LW` instruction is followed immediately with a RAW-dependent instruction, so we have:

	Original Loop	Unrolled Loop	Speedup
a.	$6 \times 2 = 12$	6	2
b.	$(4 + 1) \times 2 = 10$	9	1.1

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 first 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 first instruction in a cycle is not a mispredicted `BEQ` but the second one is (the probability of this is $(1 - p) \times p$), we waste two cycles. Without these mispredictions, we would be able to execute two instructions per cycle. We have:

	CPI
a.	$0.5 + 0.05 \times 2.5 + 0.95 \times 0.05 \times 2 = 0.720$
b.	$0.5 + 0.01 \times 2.5 + 0.99 \times 0.01 \times 2 = 0.545$

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 first 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	Speedup
a.	0.720	$0.5 + 0.05 \times 2.5 + 0.95 \times 0.05 \times 2 + 0.20 \times 0.20 \times 2 = 0.800$	1.11
b.	0.545	$0.5 + 0.01 \times 2.5 + 0.99 \times 0.01 \times 2 + 0.04 \times 0.04 \times 2 = 0.548$	1.01

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	Speedup
a.	0.720	$0.5 + 0.05 \times 2.5 + 0.95 \times 0.05 \times 2 + 0.65 \times 0.65 \times 1 = 1.143$	1.59
b.	0.545	$0.5 + 0.01 \times 2.5 + 0.99 \times 0.01 \times 2 + 0.75 \times 0.75 \times 1 = 1.107$	2.03

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	Speedup
a.	0.720	0.5	1.44
b.	0.545	0.5	1.09

4.30.5 The CPI with perfect branch prediction is now 0.25 (four instructions per cycle). A branch misprediction in the first 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	Speedup
a.	$0.25 + 0.05 \times 2.75 + 0.95 \times 0.05 \times 2.5 + 0.95^2 \times 0.05 \times 2.25 + 0.95^3 \times 0.05 \times 2 = 0.694$	0.25	2.77
b.	$0.25 + 0.01 \times 2.75 + 0.99 \times 0.01 \times 2.5 + 0.99^2 \times 0.01 \times 2.25 + 0.99^3 \times 0.01 \times 2 = 0.344$	0.25	1.37

The speedup 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 vs. 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	Speedup
a.	$0.25 + 0.05 \times 20.75 + 0.95 \times 0.05 \times 20.5 + 0.95^2 \times 0.05 \times 20.25 + 0.95^3 \times 0.05 \times 20 = 4.032$	0.25	16.13
b.	$0.25 + 0.01 \times 20.75 + 0.99 \times 0.01 \times 20.5 + 0.99^2 \times 0.01 \times 20.25 + 0.99^3 \times 0.01 \times 20 = 1.053$	0.25	4.21

We observe huge speedups 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 vs. 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 an `LW` instruction is immediately followed by a dependent instruction. We have:

	CPI
a.	(12 + 3)/12 = 1.25
b.	(9 + 2)/9 = 1.22

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: mov -4(esp), edx`
	`I3: add (edi,eax,4),edx`
	`I4: mov edx, -4(esp)`
	`I5: mov -4(esp),eax`
	`I6: cmp 0, (edi,eax,4)`
	`I7: jne Label`
		
		1 (eax from I1)
		
		2 (read from I4)
		2 (eax from I6)
			(7 + 5)/7 = 1.71

b.	`I1: add 4, edx`
	`I2: mov (edx), eax`
	`I3: add 4(edx), eax`
	`I4: add 8(edx), eax`
	`I5: mov eax, -4(edx)`
	`I6: test edx, edx`
	`I7: jl Label`
		
		2 (edx from I2)
			(7 + 2)/7 = 1.29

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.	15/7 = 2.14
b.	11/7 = 1.57

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 r2,-4(sp)`
	`I2: lw r3,-4(sp)`
	`I3: sll r2,r2,2`
	`I4: add r2,r2,r4`
	`I5: lw r2,0(r2)`
	`I6: add r3,r3,r2`
	`I7: sw r3,-4(sp)`
	`I8: lw r2,-4(sp)`
	`I9: sll r2,r2,2`
	`I10: add r2,r2,r4`
	`I11: lw r2,0(r2)`
	`I12: bne r2,zero,Label`
		I6 stalls, and all subsequent instructions have dependences so this stall remains.
		
		I9 stalls, but we can do I2 from the next iteration instead. However, this makes I3 stall and we can't eliminate that stall.
		
		I12 stalls and all subsequent instructions that remain have dependences so this stall remains.
			(12 + 3)/12 = 1.25

b.	`I1: addi r4,r4,4`
	`I2: lw r3,0(r4)`
	`I3: lw r2,4(r4)`
	`I4: add r2,r2,r3`
	`I5: lw r3,8(r4)`
	`I6: add r2,r2,r3`
	`I7: sw r2,-4(r4)`
	`I8: slt r1,r4,zero`
	`I9: bne r1,zero,Label`
		I4 stalls, but we can do I8 instead.
		
		I6 stalls, and all remaining subsequent instructions have dependences so this stall remains.
			(9 + 1)/9 = 1.11

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 t2,-4(sp)`
	`I3: sll t3,t1,2`
	`I4: add t4,t3,r4`
	`I5: lw t5,0(t4)`
	`I6: add r3,t2,t5`
	`I7: sw r3,-4(sp)`
	`I8: lw t6,-4(sp)`
	`I9: sll t7,t6,2`
	`I10: add t8,t7,r4`
	`I11: lw r2,0(t8)`
	`I12: bne r2,zero,Label`
		I6 stalls, and all subsequent instructions have dependences. Note that I8 reads what I7 wrote to memory, so these instructions are still dependent.
		
		I9 stalls, but we can do I1 from the next iteration instead.
		
		I12 stalls, but we can do I2 from the next iteration instead.
			(12 + 1)/12 = 1.08

b.	`I1: addi r4,r4,4`
	`I2: lw t1,0(r4)`
	`I3: lw t2,4(r4)`
	`I4: add t3,t2,t1`
	`I5: lw r3,8(r4)`
	`I6: add r2,t3,r3`
	`I7: sw r2,-4(r4)`
	`I8: slt r1,r4,zero`
	`I9: bne r1,zero,Label`
		This loop can now execute without stalls. I4 would stall, but we can do I5 instead. After I5 we execute I4, so I6 no longer stalls.	9/9 = 1

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,-4(sp)`
	`I3: sll t3,t1,2`
	`I4: add t4,t3,r4`
	`I5: lw t5,0(t4)`
	`I6: add t6,t2,t5`
	`I7: sw t6,-4(sp)`
	`I8: lw t7,-4(sp)`
	`I9: sll t8,t7,2`
	`I10: add t9,t8,r4`
	`I11: lw t10,0(t9)`
	`I12: bne t10,zero,Label`
		I6 stalls, and all subsequent instructions have dependences. Note that I8 reads what I7 wrote to memory, so these instructions are still dependent.
		
		I9 would stall, but we can do I1 from the next iteration instead.
		
		I12 would stall, but we can do I2 from the next iteration instead.
			(12 + 1)/12 = 1.08

b.	`I1: addi t1,t1,4`
	`I2: lw t2,0(t1)`
	`I3: lw t3,4(t1)`
	`I4: add t4,t3,t2`
	`I5: lw t5,8(t1)`
	`I6: add t6,t4,t5`
	`I7: sw t6,-4(t1)`
	`I8: slt t7,t1,zero`
	`I9: bne t7,zero,Label`
		No stalls remain. I4 would stall, but we can do I5 instead. After I5 we execute I4, so I6 no longer stalls.
		
		In next iteration uses of r4 renamed to t3.
			9/9 = 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.	$0.25 \times (1 - 0.95)$	80
b.	$0.25 \times (1 - 0.99)$	400

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.	$15 \times 4 \times 0.25 = 15$
b.	$30 \times 4 \times 0.25 = 30$

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 if branch is 1st, 2 if branch is 2nd, 1 if branch is 3rd, and 0 if branch is last). We have:

	Wrong-path Instructions
a.	$(12 - 1) \times 4 \times 1.5 = 45.5$
b.	$(20 - 1) \times 4 \times 1.5 = 77.5$

4.32.4 We can compute the CPI for each processor, then compute the speedup. 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			
	Cycles	CPI	Mis-Fetched	Cycles	CPI	Speedup
a.	(45.5 + 80)/4 = 31.4	31.4/80 = 0.392	$(12 - 1) \times 8 \times 3.5 = 91.5$	(91.5 + 80)/8 = 21.4	21.4/80 = 0.268	1.46
b.	(77.5 + 400)/4 = 119.4	119.4/400 = 0.298	$(20 - 1) \times 8 \times 3.5 = 155.5$	(155.5 + 400)/8 = 69.4	69.4/400 = 0.174	1.72

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

	"Normal" CPI	"Improved" CPI	Speedup
a.	31.4/80 = 0.392	30.4/80 = 0.380	1.033
b.	119.4/400 = 0.298	118.4/400 = 0.296	1.008

4.32.6

	"Normal" CPI	"Improved" CPI	Speedup
a.	21.4/80 = 0.268	20.4/80 = 0.255	1.049
b.	69.4/400 = 0.174	68.4/400 = 0.171	1.015

Speedups 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 (speedup).

Solution 4.33

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

	Read Ports
a.	$2 \times 2 = 4$
b.	$8 \times 2 = 16$

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 speedup of:

	Speedup
a.	15/5 × 2 = 6
b.	30/5 × 8 = 48

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

	Speedup
a.	15/5 = 3
b.	30/5 = 6

4.33.4 We first 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 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 first 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/(0.10 × 0.04) = 250	250/2 = 125	8.3	8/(125 + 8.3) = 6%
b.	1/(0.10 × 0.02) = 500	500/8 = 62.5	4.4	4/(62.5 + 4.4) = 6%

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.	8.3	8.3 × 2/0.05 = 330	1/(330 × 0.10) = 3.03%
b.	4.4	4.4 × 8/0.01 = 3550	1/(3550 × 0.10) = 0.28%

The needed accuracy is 100% minus the allowed misprediction rate.

4.33.6 This problem is very similar to 4.33.5, 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.	8.3	$8.3 \times 2 = 16.5$	$1/(16.5 \times 0.10) = 60.1\%$
b.	4.4	$4.4 \times 8 = 35.5$	$1/(35.5 \times 0.10) = 28.2\%$

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 "And" 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 And 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 read two registers in the ID stage, and we also need the sign-extend unit for the Offs field in the instruction word. In the EXE stage we need an Add unit whose inputs are the Rs 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 the value we read from Rt is used as a data value for a memory write. Note that there is no WB 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.

4.34.2

a.	Assuming that the register write in WB happens in the first half of the cycle and the register reads in ID happen in the second half, we only need to forward the And result from the EX/WB pipeline register to the inputs of the And 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.	There is no need for forwarding or hazard detection in this pipeline because there are no RAW data dependences between two store instructions.

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 filed (if there is a funct field) 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 flush (convert to NOPs) the undefined 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 replace the And unit in EXE with an ALU that supports either an Add or an And. The ALUOp signal to select between these operations must be supplied by the Control unit.
b.	The two operations are identical until the end of the EXE stage. After that, the ADDI operation must store the ALU output to the Rt register, so we must add the WB stage (SW did not need it). In fact, the work of the WB stage can be done in the MEM stage, so our pipeline remains a 4-stage pipeline. Our control logic must select whether we write the value of Rt to memory (for SW) or we write the ALU result to Rt (for ADDI).

4.34.5

a.	The same forwarding logic used for AND can be used for ADD, and we still need no hazard detection.
b.	Now we need forwarding because of ADDI instructions. Assuming that the register write in WB happens in the first half of the cycle and the register read in ID happens in the second half, we need to forward the Add result of an ADDI instruction from the EX/WB pipeline register to the first (register Rs) input of the Add unit in the EXE stage of the next instruction if that next instruction depends on the ADDI. We also need to forward that same Add result to replace the Rt value that will be stored into memory by the next SW instruction, if that instruction's Rt register is the same register as the Rt (result) register of the ADDI instruction. Fortunately, we still need no hazard detection.

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 the branch execution stage. We have:

	Delay Slots Needed
a.	$10 \times 2 - 1 = 19$
b.	$15 \times 4 - 1 = 59$

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 filling them with delay-slot instructions. We compute the number of execution (non-stall) cycles between mispredictions, and the speedup as follows:

	Non-stall Cycles between Mispredictions	Stall Cycles without Delay Slots	Stall Cycles with 4 Delay Slots	Speedup Due to Delay Slots
a.	$1/(0.25 \times (1 - 0.90) \times 2) = 20$	9	7	$(20 + 9)/(20 + 7) = 1.074$
b.	$1/(0.15 \times (1 - 0.96) \times 4) = 41.7$	14	13	$(41.7 + 14)/(41.7 + 13) = 1.018$

4.35.3 For 20% of the branches we add an extra instruction, for 30% of the branches we add two extra instructions, and for 40% of the 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 `NOP`s 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	Speedup Due to Delay Slots
a.	$20 + 9 = 29$	7	$1 \times 20 \times 0.25 = 5$	$29/(20 + 7 + 5) = 0.906$
b.	$41.7 + 14 = 55.7$	13	$0.5 \times 41.7 \times 0.15 = 3.125$	$55.7/(41.7 + 13 + 3.125) = 0.963$

4.35.4

a.

```
      add r2,zero,zero    ; r1=0
Loop: beq r2,r3,End
      lb  r10,1000(r2)    ; Delay slot
      add r1,r1,r10
      beq zero,zero,Loop
      addi r2,r2,1        ; Delay slot
Exit:
```

b.

```
      add r2,zero,zero    ; r1=0
Loop: beq r2,r3,End
      lb  r10,1000(r2)    ; Delay slot
      lb  r11,1001(r2)
      sub r12 r10,r11
      add r1,r1,r12
      beq zero,zero,Loop
      addi r2,r2,2        ; Delay slot
Exit:
```

4.35.5

a.	<pre> add r2,zero,zero ; r1=0
	Loop: beq r2,r3,End
	      lb r10,1000(r2) ; Delay slot
	      nop ; 2nd delay slot
	      beq zero,zero,Loop
	      add r1,r1,r10 ; Delay slot
	      addi r2,r2,1 ; 2nd delay slot
	Exit:</pre>

b.	<pre> add r2,zero,zero ; r1=0
	Loop: beq r2,r3,End
	      lb r10,1000(r2) ; Delay slot
	      lb r11,1001(r2) ; 2nd delay slot
	      sub r12 r10,r11
	      beq zero,zero,Loop
	      add r1,r1,r12 ; Delay slot
	      addi r2,r2,2 ; 2nd delay slot
	Exit:</pre>

4.35.6 The maximum number of in-flight instructions is equal to the pipeline depth times the issue width. We have:

	Instructions in Flight	Instructions per Iteration	Iterations in Flight
a.	15 × 2 = 30	5	30/5 + 1 = 7
b.	25 × 4 = 100	7	roundUp(100/7) + 1 = 16

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 round up.

Solution 4.36

4.36.1

	Instruction	Translation
a.	`SWINC Rt,Offset(Rs)`	`SW Rt,Offset(Rs)` `ADDI Rs,Rs,4`
b.	`SWI   Rt,Rd(Rs)`	`ADD tmp,Rd,Rs` `SW Rt,0(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 are 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 write this incremented value back into Rs.
b.	We can use the existing EX stage to perform this address calculation and then write to memory in the MEM stage. But we do need an additional (third) register read port because this instruction reads three registers in the ID stage, and we need to pass these three values to the EX 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:

	Speedup from ADDM Translation
a.	$1/(1 + 0.03 \times 3) = 0.92$
b.	$1/(1 + 0.05 \times 3) = 0.87$

4.36.6 Each translated `ADDM` adds the 3 stall cycles, but now half of the existing stalls are eliminated. We have:

	Speedup from ADDM Translation
a.	$1/(1 + 0.03 \times 3 - 0.12/2) = 0.97$
b.	$1/(1 + 0.05 \times 3 - 0.20/2) = 0.95$

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 never used.
b.	The read port of the data memory is never used (no load instructions).

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.	20% (1 out of 5)	20% (1 out of 5)
b.	0% (no LW)	25% (1 out of 4)

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 fields 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 speedup 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	"Speedup"
a.	200ps	90ps	110ps	250ps	20ps	670ps	250ps	2.68
b.	750ps	300ps	300ps	500ps	50ps	1900ps	750ps	2.53

Note that this speedup is significantly lower than 5, which is the "ideal" speedup 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	“Speedup”
a.	200ps	90ps	110ps	0ps	400ps	200ps	2.00
b.	750ps	300ps	300ps	0ps	1135ps	750ps	1.80

Note that the “ideal” speedup from pipelining is now 4 (we removed the MEM stage), and the actual speedup 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 × 670 = \$134	0.8 × 250ps = 200ps	\$50 (MEM)
b.	0.2 × 1900 = \$380	0.8 × 750ps = 600ps	\$150 (IF)

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.	140pJ + 2 × 70ps + 60pJ = 340pJ
b.	70pJ + 2 × 40pJ + 40pJ = 190pJ

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.	140pJ + 2 × 70pJ + 60pJ + 140pJ = 480pJ
b.	70pJ + 2 × 40pJ + 40pJ + + 90pJ = 280pJ

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, an `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.	140pJ + 2 × 70pJ + 60pJ + 140pJ = 480pJ	70pJ	14.6%
b.	70pJ + 2 × 40pJ + 40pJ + + 90pJ = 280pJ	40pJ	14.3%

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.	250ps (D-Mem in MEM stage)	No change (150ps + 90ps<250ps)
b.	750ps (I-Mem in IF stage)	800ps (Ctl then Regs in ID stage)

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 stalls). This change does not affect clock cycle time because the clock cycle time must already allow enough time for memory to be read in the MEM stage. It does affect energy: a memory read occurs in every cycle instead of only in cycles when a load instruction is in the MEM stage.

4.38.6

	I-Mem Active Energy	I-Mem Latency	Clock Cycle Time	Total I-Mem Energy	Idle Energy %
a.	140pJ	200ps	250ps	140pJ + 50ps × 0.1 × 140pJ/200ps = 143.5pJ	3.5pJ/143.5pJ = 2.44%
b.	70pJ	750ps	750ps	70pJ	0%

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 the 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.	0.75	350ps	2.86GHz	2.14×10^9
b.	0.80	220ps	4.55GHz	3.64×10^9

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 five stages. We have:

	Clock Frequency	Energy per Cycle (in pJ)	Power (W)
a.	2.86GHz	$100 + 45 + 50 + 0.30 \times 150 + 0.45 \times 50 = 262.5$	0.75
b.	4.55GHz	$75 + 45 + 100 + 0.45 \times 100 + 0.50 \times 35 = 282.5$	1.28

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.	350ps	100ps	0ps	200ps	50ps	150ps
b.	220ps	20ps	50ps	0ps	10ps	70ps

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.	350/250	350/350	350/150	350/300	350/200	0.54
b.	220/200	220/170	220/220	220/210	220/150	1.17

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 4.39.4. We get:

	X for IF	X for ID	X for EX	X for MEM	X for WB	New Power (W)
a.	385/250	385/350	385/150	385/300	385/200	0.45
b.	242/200	242/170	242/220	242/210	242/150	0.97

4.39.6 The X factor for each stage is the same as in 4.39.6, 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.	0.31	0.75	58.7%
b.	0.81	1.28	36.7%

5 Solutions

Solution 5.1

5.1.1

a.	no solution provided
b.	no solution provided

5.1.2

a.	no solution provided
b.	no solution provided

5.1.3

a.	no solution provided
b.	no solution provided

5.1.4

a.	no solution provided
b.	no solution provided

5.1.5

a.	no solution provided
b.	no solution provided

5.1.6

a.	no solution provided
b.	no solution provided

Solution 5.2

5.2.1 4

5.2.2

a.	I, J
b.	B[I][0]

5.2.3

a.	A[I][J]
b.	A[J][I]

5.2.4

a.	$3596 = 8 \times 800/4 \times 2 - 8 \times 8/4 + 8000/4$
b.	$3186 = 8 \times 800/4 \times 2 - 8 \times 8/4 + 8/4$

5.2.5

a.	I, J
b.	I, J, B(I, 0)

5.2.6

a.	A(J, I)
b.	A(I, J), A(J, I), B(I, 0)

Solution 5.3

5.3.1

a.	no solution provided
b.	no solution provided

5.3.2

a.	no solution provided
b.	no solution provided

5.3.3

a.	No solution provided
b.	no solution provided

5.3.4

a.	no solution provided
b.	no solution provided

5.3.5 no solution provided

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

Solution 5.4

5.4.1

a.	8
b.	16

5.4.2

a.	32
b.	64

5.4.3

a.	1 + (22/8/32) = 1.086
b.	1 + (20/8/64) = 1.039

5.4.4 3

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

5.4.5 0.25

5.4.6 <Index, tag, data>

$<000001_2, 0001_2, \text{mem}[1024]>$
$<000001_2, 0011_2, \text{mem}[16]>$
$<001011_2, 0000_2, \text{mem}[176]>$
$<001000_2, 0010_2, \text{mem}[2176]>$
$<001110_2, 0000_2, \text{mem}[224]>$
$<001010_2, 0000_2, \text{mem}[160]>$

Solution 5.5

5.5.1

a.	no solution provided
b.	no solution provided

5.5.2

a.	no solution provided
b.	no solution provided

5.5.3

a.	no solution provided
b.	no solution provided

5.5.4

a.	no solution provided
b.	no solution provided

5.5.5

a.	no solution provided
b.	no solution provided

5.5.6

a.	no solution provided
b.	no solution provided

Solution 5.6

5.6.1

no solution provided

5.6.2

no solution provided

5.6.3 With next-line prefetching, miss rate will be near 0%.

5.6.4

a.	no solution provided
b.	no solution provided

5.6.5

a.	no solution provided
b.	no solution provided

5.6.6

a.	no solution provided
b.	no solution provided

Solution 5.7

5.7.1

a.	P1	1.52 GHz
	P2	1.11 GHz
b.	P1	926 MHz
	P2	495 MHz

5.7.2

a.	P1	6.31 ns	9.56 cycles
	P2	5.11 ns	5.68 cycles
b.	P1	3.47 ns	3.21 cycles
	P2	4.07 ns	2.02 cycles

5.7.3

a.	P1	12.64 CPI	8.34 ns per inst	P2
	P2	7.36 CPI	6.63 ns per inst	
b.	P1	4.01 CPI	4.33 ns per inst	P1
	P2	2.38 CPI	4.81 ns per inst	

5.7.4

a.	6.50 ns	9.85 cycles	Worse
b.	3.84 ns	3.25 cycles	Worse

5.7.5

a.	13.04
b.	4.06

5.7.6 no solution provided

Solution 5.8

5.8.1

a.	no solution provided
b.	no solution provided

5.8.2

a.	no solution provided
b.	no solution provided

5.8.3

a.	no solution provided
b.	no solution provided

5.8.4

a.	no solution provided
b.	no solution provided

5.8.5

a.	no solution provided
b.	no solution provided

5.8.6

a.	no solution provided
b.	no solution provided

Solution 5.9

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

5.9.1 32 KB.

5.9.2 Still 32 KB.

5.9.3 64 KB. Because the disk bandwidth grows much faster than seek latency, future paging cost will be closer to constant, thus favoring larger pages.

5.9.4 1987/1997/2007: 205/267/308 seconds (or roughly five minutes).

5.9.5 1987/1997/2007: 51/533/4935 seconds (or 10 times longer for every 10 years).

5.9.6 (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.

Solution 5.10

5.10.1

a.	no solution provided
b.	no solution provided

5.10.2

a.	no solution provided
b.	no solution provided

5.10.3

a.	no solution provided
b.	no solution provided

5.10.4

a.	no solution provided
b.	no solution provided

5.10.5

a.	no solution provided
b.	no solution provided

5.10.6

a.	no solution provided
b.	no solution provided

Solution 5.11

5.11.1

a.	no solution provided
b.	no solution provided

5.11.2

a.	no solution provided
b.	no solution provided

5.11.3

a.	no solution provided
b.	no solution provided

5.11.4 TLB initialization, or process context switch.

5.11.5 TLB miss. When most missed TLB entries are cached in processor caches.

5.11.6 Write protection exception.

Solution 5.12

5.12.1

a.	0 hits
b.	3 hits

5.12.2

a.	1 hit
b.	3 hits

5.12.3

a.	1 hit or fewer
b.	4 hits or fewer

5.12.4 5.12.4 Any address sequence is fine so long as the number of hits is correct.

a.	1 hit
b.	4 hits

5.12.5 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.

5.12.6 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.

Solution 5.13

5.13.1 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.

5.13.2

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

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

5.13.3

Shadow page table: page fault rate

NPT: TLB miss rate

5.13.4

Shadow page table: 1.03

NPT: 1.04

5.13.5 Combining multiple page table updates

5.13.6 NPT caching (similar to TLB caching)

Solution 5.14

5.14.1

a.	no solution provided
b.	no solution provided

5.14.2

a.	no solution provided
b.	no solution provided

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

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

Solution 5.15

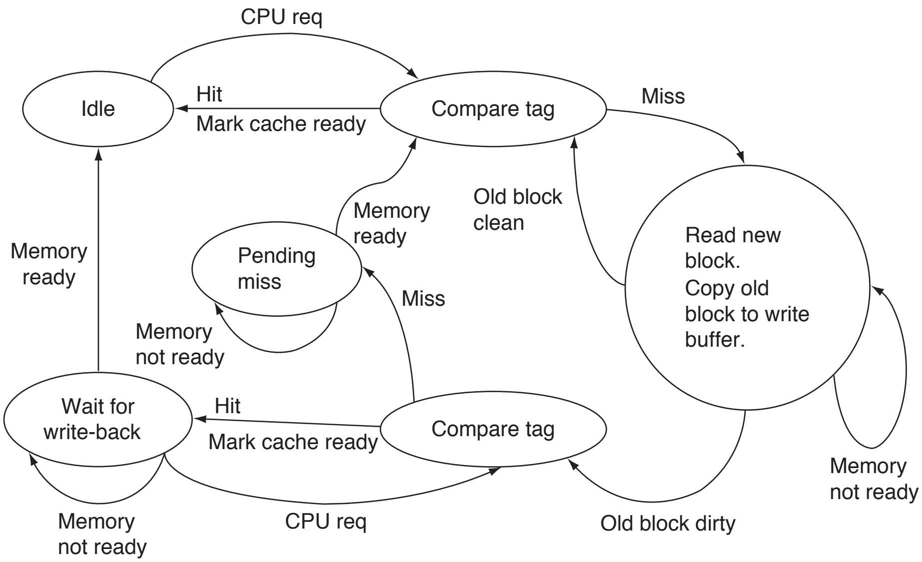
5.15.1 The cache should be able to satisfy the request since it is otherwise idle when the write buffer is writing back to memory. If the cache is not able to satisfy hits while writing back from the write buffer, the cache will perform little or no better than the cache without the write buffer, since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

5.15.2 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.

5.14.3 Correct solutions should exhibit the following features:

1. The memory read should come before memory writes.
2. The cache should signal "Ready" to the processor before completing the write.

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



Solution 5.16

5.16.1

a.	no solution provided
b.	no solution provided

5.16.2 no solution provided

5.16.3

a.	no solution provided
b.	no solution provided

5.16.4

a.	no solution provided
b.	no solution provided

5.16.5

a.	no solution provided
b.	no solution provided

5.16.6 Write-through, non-write allocate simplifies the most.

Solution 5.17

5.17.1

a.	no solution provided
b.	no solution provided

5.17.2

a.	no solution provided
b.	no solution provided

5.17.3

a.	no solution provided
b.	no solution provided

5.17.4

a.	no solution provided
b.	no solution provided

5.17.5

a.	no solution provided
b.	no solution provided

5.17.6

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

Solution 5.18

5.18.1

a.	`srcIP` and `refTime` fields. 2 misses per entry.
b.	`srcIP` and `browser` fields. 1 miss per entry.

5.18.2

a.	Group the `srcIP` and `refTime` fields into a separate array.
b.	Split the `srcIP` into a separate array; have a hash table on the browser field.

5.18.3

a.	`peak_hour (int status); // peak hours of a given status`
	Group `srcIP`, `refTime`, and `status` together.

b.	`topK_sourceIP (int hour);` Group the `srcIP` and `refTime` fields into a separate array, and a browser hash table.

5.18.4

a.	no solution provided
b.	no solution provided

5.18.5

a.	no solution provided
b.	no solution provided

5.18.6

a.	apsi/mesa/ammp/mcf all have such examples.
b.	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.	Auto Pilot	Keypad – Input, Human Display – Output, Human Alarms – Output, Human Control Surfaces – I/O, Machine
b.	Automated Thermostat	Keypad – Input, Human Control Signals – Output, Machine

6.1.2

a.	Auto Pilot	Keypad – 0.0001 Mbit/sec Display – 800 Mbit/sec Alarms – 0.00001 (highly variable) Mbit/sec Control Surfaces – 0.1 (highly variable) Mbit/sec
b.	Automated Thermostat	Keypad – 0.0001 Mbit/sec Control Signals – 0.00001 Mbit/sec

6.1.3

a.	Auto Pilot	Keypad – Operation Rate Display – Data Rate Alarms – Operation Rate Control Surfaces – Operation Rate for most applications
b.	Automated Thermostat	Keypad – Operation Rate Control Signals – Operation Rate

Solution 6.2

6.2.1

a.	1096 days	26,304 hours
b.	2558 days	61,392 hours

6.2.2

a.	0.9990875912%
b.	0.9988272088%

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 file systems and other data can take significant time. Although a drive manufacturer will not include this time in their statistics, it is certainly a part of replacing a disk.

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, the specific value of MTTR is not significant.

Solution 6.3

6.3.1

a.	14.011 ms
b.	10.025 ms

6.3.2

a.	14.022
b.	10.05

6.3.3 The dominant factor for all disks seems to be the average seek time, although RPM would make a significant 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.	No	An aircraft control system will process frequent requests for small amounts of information. Increasing the sector size will decrease the rate at which requests can be processed.
b.	No	A phone switch processes frequent requests for small data elements. Increasing sector size will potentially reduce performance.

6.4.2

a.	No	An aircraft control system is not typically I/O limited. Faster access to disk may be useful in some situations, but not normal operation.
b.	No	A phone switch should not be I/O limited. Faster access to disk may be useful, but may improve performance in limited scenarios.

6.4.3

a.	No	Failure in an aircraft control system is not tolerable. Increasing disk failure rate for faster data access is not acceptable.
b.	No	Failure in a phone switch is not tolerable. Increasing disk failure rate for faster data access is not acceptable.

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 first technology with potential to catch hard disk.

6.5.2

a.	No	Increased drive performance is not an issue in an aircraft controller.
b.	No	Increased drive performance is not an issue in a phone switch.

6.5.3

a.	No	
b.	No	

Solution 6.6

6.6.1 Note that some of the specified FLASH memories are controller limited. This is to convince you to think about the system rather than simply the FLASH memory.

a.	9.77 ms
b.	10.85 ms

6.6.2 Note that some of the specified FLASH memories are controller limited. This is to convince you to think about the system rather than simply the FLASH memory.

a.	4.89 ms
b.	5.43 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. The printer is electrically distant from the CPU.
b.	Asynchronous. Scanner inputs are relatively infrequent in comparison to other inputs. The scanner itself is electrically distant from the CPU.

6.7.2 For all devices in the table, problems with long, synchronous busses are the same. Specifically, long synchronous busses 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 significant 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 busses 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 due to distance from the CPU and low bandwidth requirements. FireWire would not be as appropriate due to its daisy chaining implementation.
b.	PCI due to higher throughput. No need for hot swap capabilities and the device will be close to the CPU.

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 FireWare, SATA devices are hot swappable.

6.8.3

Bus Type	Drawbacks
PCI	The parallel bus used to transmit data limits the length of the bus. Having a fixed number of control lines limits the number of devices on the bus. The trade-off is speed. PCI busses 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 busses 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. Interface may be handled by polling, but not control or sensor inputs.
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.	Aircraft surfaces generate interrupts caused by movements. Controller generates signals back to control surfaces. User displays can be managed by either polling or interrupts.
b.	Polling is okay.

6.9.3 Basically, each interface is designed in a similar way with memory locations identified for inputs and outputs associated with devices.

a.	The autopilot is an input/output device. It inputs 32 single word values from various sensors on control surfaces and generates 32 single word values as control signals to actuators. Status for 32 potential alarm values is stored in one word while four words store navigational information.
b.	An automated thermostat is a simple device, but it has both input and output functions. It uses a keypad for communication to the user and on/off outputs to communicate with a furnace and air conditioner. The keypad memory should hold values input by toggle switches and numeric entries. The on/off outputs can be mapped to single bits in memory.

6.9.4

a.	The autopilot is an input/output device that requires significant I/O with a user and control surfaces. User I/O can be handled by commands that fetch input information. Similarly, control surfaces can be controlled by issuing individual commands or issuing commands with state for several sensors.
b.	An automated thermostat is a simple device, but it has both input and output functions. It uses a keypad for communication to the user and on/off outputs to communicate with a furnace and air conditioner. The keypad memory should hold values input by toggle switches and numeric entries. The on/off outputs can be mapped to single bits in memory.

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 can be 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 handling of the current interrupt that is higher priority. The status register is saved to assure that any lower priority interrupts that have been detected are handled when the status register is restored following handling of the current interrupt.

6.10.2 Lower numbers have higher interrupt priorities.

a.	Ethernet Controller Data: 2	Mouse Controller: 3	Reboot: 1
b.	Mouse Controller: 3	Power Down: 2	Overheat: 1

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. Specifically, 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 this solution requires storing information on the stack, in registers, in a dedicated memory area, or some combination of the three. Providing hardware support removes the burden of storing program state from the operating system. Specifically, program state information need not be pulled from the CPU and stored in memory.

This is essentially the same as 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 first 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.	No. The dataflow back and forth from a mouse is insignificant.
b.	Possibly. One thought is the Ethernet controller handles significant amounts of data. However, that data is typically in relatively small packets. Depending on the functionality performed by the controller, it may or may not make sense to have it use DMA.

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 mouse controller will not use DMA.
b.	No. The Ethernet controller will not use DMA.

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. Specifically, 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 to be accessed, but likely the next memory element being accessed.

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 sufficient physical memory will almost always solve this problem.

Solution 6.12

6.12.1

a.	Not typically, although it is possible.
b.	Yes.

6.12.2

a.	N/A
b.	No. Online chat is dominated by transactions, not the size of those transactions.

6.12.3 See the previous problem for explanations.

a.	N/A
b.	Yes.

6.12.4 Polling would be more inappropriate for applications where 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 difficult. In most situations, a mixture of the two approaches is the most pragmatic approach. Specifically, use commands to handle interactions and memory to exchange data. For transaction dominated I/O, command driven I/O will likely be sufficient.

Solution 6.13

6.13.1

a.	Large, concurrent data reads and writes.
b.	Large numbers of small, concurrent transactions.

6.13.2 Standard benchmarks help when trying to compare and contrast different systems. Ranking systems with benchmarks is generally not useful. However, understanding trade-offs 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 difficult 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, unless computations force the system to access disk frequently.
b.	No. The bottleneck in such systems is network throughput not disk I/O.

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 offline, revenue loss is immediate and customers lose confidence in the service.

6.14.3 RAID 1 maintains two complete copies of a dataset while RAID 3 maintains error correction data only. The trade-off is storage cost. RAID 1 requires two 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 definitely benefit from RAID 3. Video and sound editing may also benefit from RAID 3, but these applications are not as sensitive to availability issues as online services.

Solution 6.15

6.15.1

a.	DEE8
b.	7B25

6.15.2

a.	F030
b.	78E9

6.15.3 RAID 4 is more efficient because it requires fewer reads to generate the next parity word value. Specifically, 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 efficient.

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 efficient. For lower volume, RAID 5 will not significantly 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 3 for small numbers of disks. For two disks, there is no difference.

Solution 6.16

6.16.1

a.	8000
b.	7500

6.16.2

	16 Disks		8 Disks		4 Disks		2 Disks	
	IOPS	Bottleneck?	IOPS	Bottleneck?	IOPS	Bottleneck?	IOPS	Bottleneck?
a.	28000	No	14000	No	7000	Yes	3500	Yes
b.	14000	No	7000	Yes	3500	Yes	1750	Yes

6.16.3

	PCI Bus		DIMM		Front Side Bus	
	IOPS	Bottleneck?	IOPS	Bottleneck?	IOPS	Bottleneck?
a.	31250	No	83375	No	165625	No
b.	15625	No	41687.5	No	82812.5	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 an 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 throughput 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 find advertisements in periodicals from your professional societies or trade journals. You should be able to identify one or more candidates using the criteria identified in 6.17.1. If your reasons for selecting the server don't follow from 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 configurations. You should consider 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 configurations 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, those that 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.	9.125

6.18.2

	7 Years	10 Years
a.	26.28	189.8
b.	32.85	237.25

6.18.3 Average failure rates of the drives with longer longevity for 7 and 10 years are:

	7 Years	10 Years
a.	16.06	36.5
b.	12.775	38.325

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 first. The ratio of the number of drives replaced in the first scenario to the number replaced in the second should give us the multiple that we want:

	7 Years	10 Years
a.	1.64	5.2
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 a price impact on the company.

6.19.2 As a search engine provider paid by ad hits, throughput is critical. Most HTTP traffic 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 first 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 configuration, 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 justification 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—portability 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 identified.

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

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 benefit from these extra cores. But if we create threads to compare the N elements to the value X and perform these in parallel, then we can get ideal speedup (Y times speedup), and the comparison can be completed in the amount of time to perform a single comparison.

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

Solution 7.3

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

Version 1—17,965 cycles

Version 2—22,955 cycles

Version 3—20,959 cycles

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 speedup by eliminating many loop increment, branch and load instructions. The loop body running on node 1 would look something like this (the code is not the most efficient code sequence):

```
      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 instruction latency.

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

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 speedup 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 $1 + 2 + 4 + 8 + 16 + \ldots . \log_2(m)$ processors to obtain speedup.

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

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 find the speedup 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 speedup should be very close to 4.

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

7.6.3 In this question, we are asked how to fix this problem. The easiest way to solve the false sharing problem is to compute the elements in C by traversing the matrix across columns instead of rows (i.e., using index-j instead of index-i). These elements will be mapped to different cache lines. Then we just need to make sure we 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 × C entries = number of bytes consumed in the cache for maintaining coherence.

7.8.2 P bytes/entry × 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 "dirty". Specific to the coherency protocol assumed, on the first 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 traffic 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 first 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 first CPU is invalidated. The local write activity is the same as for the first 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 500 cycle stall every 375 cycles

7.10.2 We will encounter a 600 cycle stall every 375 cycles

7.10.3 We will encounter a 400 cycle stall every 375 cycles

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 finishes eating, she checks to see if her neighbors want to eat and are waiting. If so, then she releases the fork to one of them and lets them eat.

The difficulty is to first be able to obtain both forks without another philosopher interrupting the transition between checking and acquisition. We can implement this a number of ways, but a simple way is to accept requests for forks in a centralized queue, and give out forks based on the priority defined by being closest to the head of the queue. This provides both deadlock prevention and fairness.

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
A3	B1, B4
A1, A2	B1, B4
A1, A4	B2
A1	B3

7.12.2

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

7.12.3

FU1	FU2
A1	B1
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 processes that will compute ¼ of the sums. Assuming that memory latency is not an issue, the program should get linear speed when run on the 4 processors (there is no communication necessary between threads). If memory is being considered in the answer, then the array blocking should consider preserving spatial locality so that false sharing is not created.

7.14.2 Since this program is highly data parallel and there are no data dependencies, a 8X speedup 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.

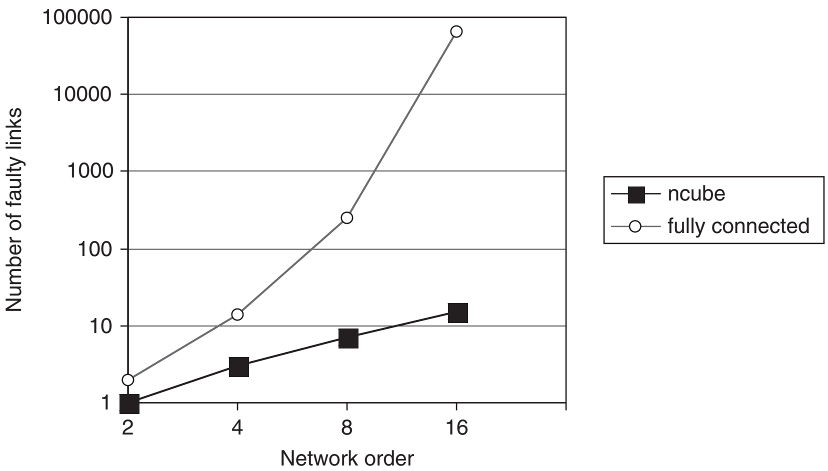
Solution 7.18

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 students become aware of these technologies. The actual bandwidth and latency values should be available right off the company websites, and will change as the technology evolves.

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 specifically

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 floating point number + (m + 1) times the size of the index, where R is the number of non-zero elements and m is the number of rows. We will assume each floating-point number is 4 bytes, and each index is a short unsigned integer that is 2 bytes.

For Matrix *X* this equals 111 bytes.

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 efficient 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 conditional jumps based on floating point registers. We would increment two counts based on the conditional compare. Finally, we could just add the two counts for the entire matrix. We would not need to use core B.

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

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 computed 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 taking place. Thus, each core must have 1.25 transactions either executing or in some amount of completion on average.

So the answers are:

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

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.