Chapter 1 Selected Solutions

1. Effective CPI = [45 x 1 + (32 + 15 + 8) x 2]/(45 + 32 + 15 + 8) = (45 + 110)/100 = 1.55
MIPS rating = f/(CPI x 10^6) = 400/1.55 = 258
Execution time = I_c x CPI x τ = 100 x 10^4 x 1.55/(400 x 10^6) = 1.55/400 = 0.003875 seconds

3. (a) MIPS rating = f/(CPI x 10^6).
Therefore, CPI = f/(MIPS x 10^6) = 1.5

(b) It is implied here that 100 – 35 = 65% of the instructions require no memory access; i.e., they execute in 1 clock cycle each.
Here, an instruction making one memory access needs 1 + 2 = 3 clock cycles and an instruction making two memory accesses needs 1 + 2 x 2 = 5 clock cycles.
Therefore, CPI = 0.65 x 1 + 0.3 x 3 + 0.05 x 5 = 1.8.
Therefore, MIPS rating = f/(CPI x 10^6) = 3000/1.8 = 1667.

4. (a) CPI = 0.6 x 1 + 0.18 x 2 + 0.12 x 4 + 0.1 x 8 = 2.24

(b) MIPS rating = f/(CPI x 10^6) = 400/2.24 = 179

5. False, True, False, False, True

6. MIPS rating = (number of instructions executed/time taken)/10^6

Accordingly, the following table has the results:

	MIPS(A)	MIPS(B)	MIPS(C)
Program 1	1000	100	50
Program 2	1	10	50
Program 3	2	1	20
Program 4	10	1.25	10

The following inferences seem reasonable:

1. The architecture of computer A is most suited to Program 1, probably exploiting to the fullest extent the inherent parallelism of the program. To a smaller extent, this is true also for Computer B.
2. However, Computers A and B do not perform too well on Programs 2, 3 and 4.
3. In terms of overall performance, Computer C outperforms the other two.

8. (a) We have a total of four Loads/Stores, one Multiply and one Add. Therefore, clock cycles needed for one iteration = 4 x 4 + 8 + 2 = 26.
Therefore clock cycles needed on a single processor, ignoring other delays = 64 x 26 = 1664.

(b) On an SIMD system with 64 PEs, clock cycles needed = 26.

(c) Speedup achieved = 64.

9. The argument is based on the time taken to broadcast a result from one PE to all other PEs. In one cycle, a result goes from one PE to a second PE; in two cycles, it reaches 2 x 2 = 4 PEs. Thus in k cycles, it reaches 2^k PEs. Therefore to share a result amongst N processors takes log(N) cycles, which is a standard result.

An EREW system, by broadcasting its results in this manner after every instruction, overcomes the limitation of not having the concurrent read and write capability of a CRCW system. Of course, we must assume the same number of PEs in both (= N).

Thus, any program running on the EREW system can be no more that O(log(N)) times slower than the same program running on a CRCW system.

10. This problem is a carry-over from the first edition of the book. In view of the very sophisticated integer multiplier algorithms now available, the N^2 term should now probably be replaced by a log(N) term. Integer multiplication on modern processors takes one cycle and, with hierarchical carry look-ahead, the asymptotic bound would grow as log(N).

However, since I have not studied in full depth today's best integer algorithms, may I suggest omitting Exercise 10 in its present form.

13. Assume that N is a power of two: $N = 2^k$.

In the first iteration, values are compared pair-wise between processors 0 and 1, 2 and 3, 4 and 5, … and so on between 2^k - 2 and 2^k - 1. After each comparison, the larger of the two numbers is made available In the lower numbered PE.

In the second iteration, values are compared pair-wise between processors 0 and 2, 4 and 6, 8 and 10, … and so on between 2^k- 4 and 2^k - 2. Again, after each comparison, the larger of the two numbers is made available in the lower numbered PE.

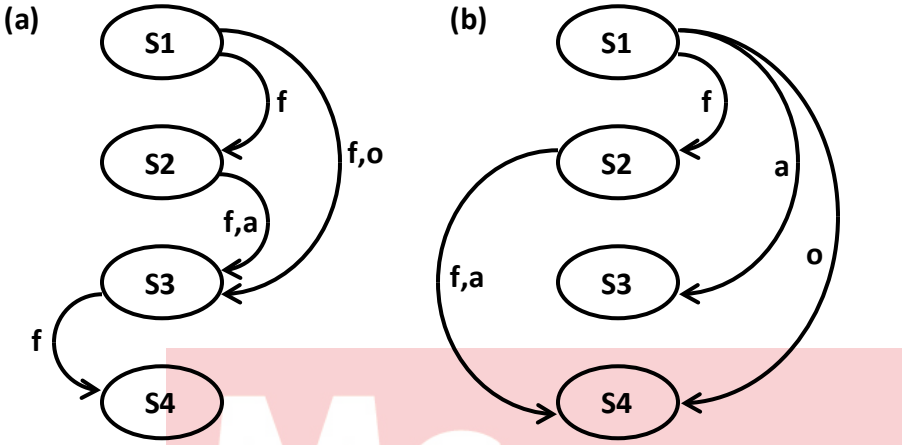
After $k = \log_2(N)$ iterations, the largest number will be in processor 0.

The second part of the exercise is not clear. A forward reference to Section 13.2.3 on Parallel Algorithms may be helpful at this stage.

14. A good and simple way to handle this and similar exercises is by making a forward reference to Brent's theorem (page 648). Basically, a one-semester course on Computer Architecture today cannot also at the same time be a course on Parallel Algorithms, even though some amount of cross-referencing is inevitable between the two.

Chapter 2 Selected Solutions

4.



Note: Because of the multiple types of data dependence present between the statements, slightly different notation is used here as compared to that in the text. Letters 'f', 'a' and 'o' indicate flow dependence, anti-dependence and output dependence, respectively.

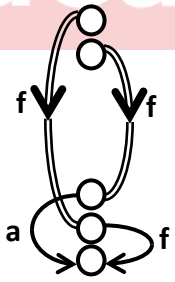
(c) Statements of two successive iterations are shown in the diagram. Value 'I + 1' is substituted everywhere for value 'I' in the lower three statements.

With this 'trick', the index expression gives information about dependence between successive statements (as shown in double line).

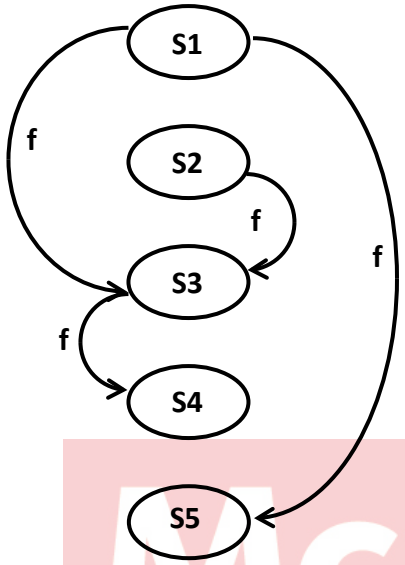
Dependences within one iteration can be shown either as a set of three nodes. In the diagram, single lines show the dependences in the lower three nodes.

```
A(I) = B(I- 2) + C(I-1)
B(I-1) = A(I-1) * K
C(I-1) = B(I-1) – 1


A(I+1) = B(I-1) + C(I)
B(I) = A(I) * K
C(I) = B(I) – 1
```

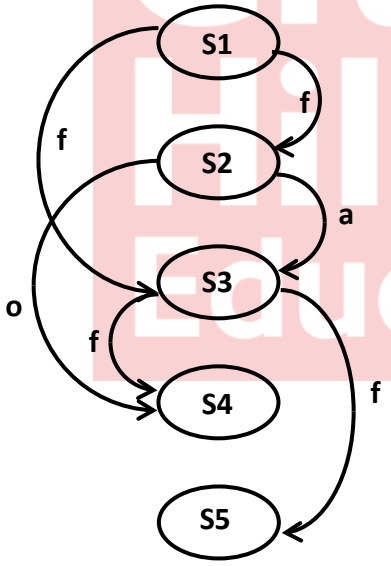


5. (a)



(b) The only possible resource dependence will be caused by the single LOAD/STORE unit available.

(c)



6. Table showing input and output variables of the five statements:

Statement index	Statement	I_J	O_K
1	A = B + C	{ B, C }	{ A }
2	C = B x D	{ B, D }	{ C }
3	S = 0		{ S }
4	DO I = A, 100 S = S + X(I) END DO	{ A, X, S }	{ S }
5	IF (S.GT.1000) C = C x 2	{ S, C }	{ C }

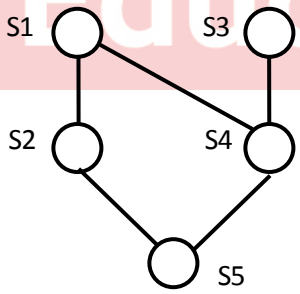
Table with relevant information about the sets:

Statement index	1	2	3	4	5
1	NA	$I_1 \sqcap O_2 \neq \Phi$	Φ	$O_1 \sqcap I_4 \neq \Phi$	$I_1 \sqcap O_5 \neq \Phi$
2	NA	NA	Φ	Φ	$I_5 \sqcap O_2 \neq \Phi$ $O_2 \sqcap O_5 \neq \Phi$
3	NA	NA	NA	$I_4 \sqcap O_3 \neq \Phi$ $O_3 \sqcap O_4 \neq \Phi$	$I_5 \sqcap O_3 \neq \Phi$
4	NA	NA	NA	NA	$I_5 \sqcap O_4 \neq \Phi$
5	NA	NA	NA	NA	NA

From this, applying Bernstein's conditions, we get:

S1 || S3, S2 || S3 and S2 || S4

Parallelized program:

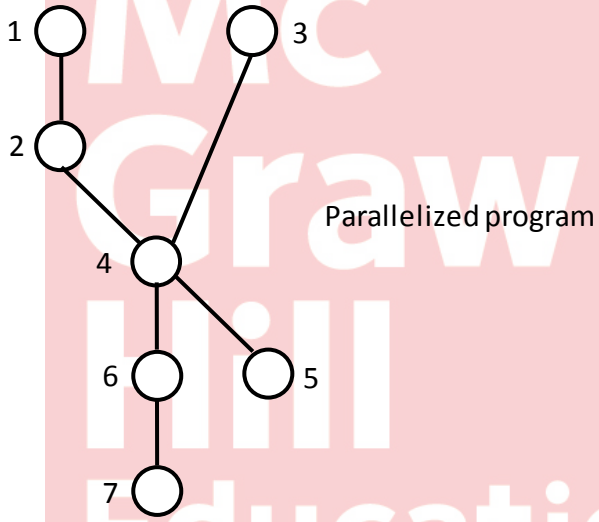


NOTE: In fact most of the latent parallelism in this program fragment is in the loop. But exploiting that parallelism requires splitting the loop.

7. Table with relevant information about the I_j and O_k sets:

[Note: Table shows only whether (a) Any of the three intersections are non-null: **≠ Φ**, or (b) All three intersections are null: **Φ**]

Statement index	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	NA	**≠ Φ**	**Φ**	**≠ Φ**	**Φ**	**≠ Φ**	**≠ Φ**
2	NA	NA	**Φ**	**≠ Φ**	**Φ**	**≠ Φ**	**Φ**
3	NA	NA	NA	**≠ Φ**	**Φ**	**Φ**	**Φ**
4	NA	NA	NA	NA	**≠ Φ**	**≠ Φ**	**≠ Φ**
5	NA	NA	NA	NA	NA	**Φ**	**Φ**
6	NA	NA	NA	NA	NA	NA	**≠ Φ**
7	NA	NA	NA	NA	NA	NA	NA



Using **Cobegin** and **Coend**:

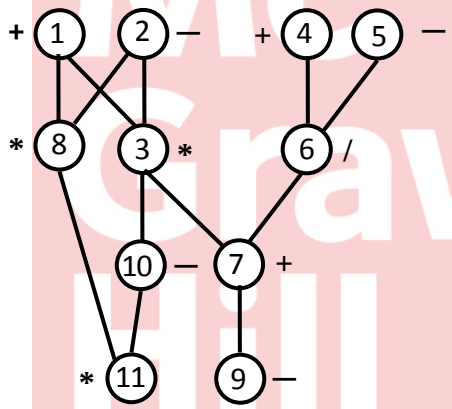
Cobegin
{ S1; S2 }
{S3}
Coend;
S4;
Cobegin
{ S5 }
{ S6; S7 }
Coend

8. (a) To achieve parallelism, we use a separate temporary register for every parenthesized sub-expression appearing on the right in the given program. This gives:

P1: R1 = A + B
P2: R2 = A - B
P3: X = R1 * R2
P4: R3 = C + D
P5: R4 = C - D
P6: Y = R3 / R4
P7: Z = X + Y
P8: A = E * F
P9: Y = E – Z
P10: R5 = X – F
P11: B = R5 * A

[It is not yet determined whether one of registers R1-R4 can be used in place of R5 in instructions P10 and P11. Even if that is done, the code length remains the same.]

(b)



Parallelized program

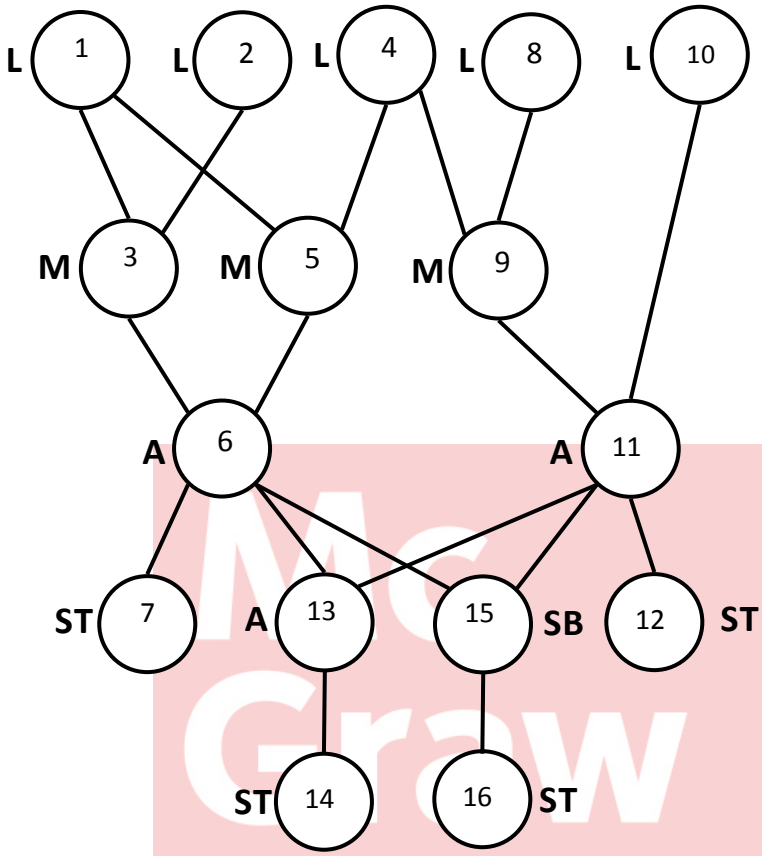
[Now it is seen that one of registers R1 or R2 can be used in place of R5.]

(c) Operations 1 to 11 are shown in clock cycles (CC) below:

ADD unit 1	ADD unit 2	MULTIPLY unit	DIVIDE unit
1 (CC = 1)	2 (CC = 1)		
4 (CC = 2)	5 (CC = 2)	3 (CC = 2 - 4)	
		8 (CC = 5 -7)	6 (CC = 3 - 20)
10 (CC = 5)	7 (CC = 21)		
	9 (CC = 22)	11 (CC = 8 - 10)	

Last operation is ADD in clock cycle 22. That is: Total time = 22 cycles, dominated by the divide operation.

9.



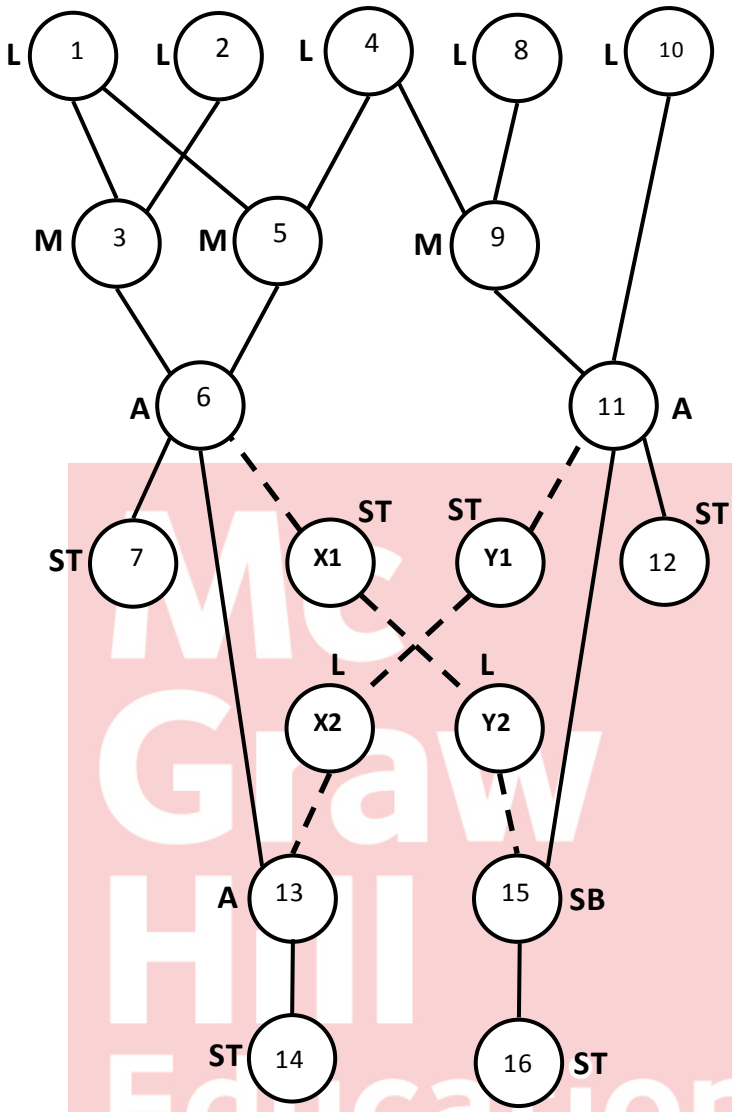
Parallelized program

L = Load, ST = Store
M = Mul, A = Add
SB = Sub

(b)

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
Load/Store	L 1	L 2	L 4	L 8	L 10	ST 7	ST 12	ST 14	ST 16
Add/Sub					A 6	A 11	A 13	SB 15	
Multiply			M 3	M 5	M 9				

10. (a)



PARALLELIZED PROGRAM

L = Load, ST = Store
M = Mul, A = Add
SB = Sub

X and Y designate the two processors.

See X and Y schedules in the given table.

(b)

		CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
X	Load/Store	L 1	L 2	L 4			ST X1	L X2	ST 7	ST 14
	Add/Sub					A 6			A13	
	Multiply			M 3	M 5					
Y	Load/Store	L 4	L 8	L 10		ST Y1	ST12	L Y2		ST 16
	Add/Sub				A 11				SB 15	
	Multiply			M 9						

NO ADVANTAGE seen with two processors, because of the many LOAD/STORE ops. Two clock cycles spent to pass data between registers of X and Y. Other units idle.

11. (a) With figure of merit as defined:

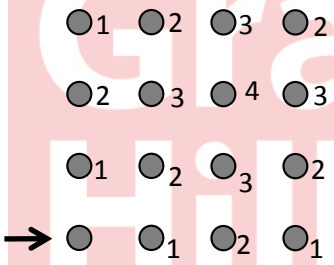
	node degree *d*	diameter *D*	number of links *L*	figure of merit
3D TORUS	6	6	192	1/6912
BINARY 6CUBE	6	6	192	1/6912
4D CCC	3	8	128	1/3072

Ranking can be seen in the right column.

(b) Note that all these networks are *symmetrical*.

[1] Consider the 3D TORUS.

In the plane closest to observer, consider the number of nodes at distance 1, 2, 3 and 4, from the node at bottom left (arrow), keeping in mind 'wrap-around' links.



These node counts are summarized in the given table.

Distance	# of nodes
1	4
2	6
3	4
4	1

Now we look in the third dimension ('into the paper'). Counts will apply as in one row — with 'wrap-around' (see bottom row above). Therefore:

# nodes at distance 1:	#(1,0) + #(0,1) = 4 + 2	= 06
# nodes at distance 2:	#(2,0) + #(1,1) + #(0,2) = 6 + 8 + 1	= 15
# nodes at distance 3:	#(3,0) + #(2,1) + #(1,2) = 4 + 12 + 4	= 20
# nodes at distance 4:	#(4,0) + #(3,1) + #(2,2) = 1 + 8 + 6	= 15
# nodes at distance 5:	#(4,1) + #(3,2) = 2 + 4	= 06
# nodes at distance 6:	#(4,2)	= 01

Notation #(J,K) means distance J in the plane shown, and distance K 'into the paper'.

The above numbers agree with the formula ${}_6\mathbf{C}_i$, for i = 1..6. The reason seems to be that a 3D TORUS of 64 nodes is isomorphic to a **binary 6-cube**.

Total probability of sending message to any node must be = 1. The denominator of the probability formula given in part (b) can be determined thereby, giving value:
6 x 6 + 15 x 5 + 20 x 4 + 15 x 3 + 6 x 2 + 1 x 1 = 249. Therefore:

# hops i	1	2	3	4	5	6
# nodes	6	15	20	15	6	1
Prob(i)	6/249	5/249	4/249	3/249	2/249	1/249

Therefore, weighted average # of hops
= sum of column-wise products
= 1/249[1 x 6 x 6 + 2 x 15 x 5 + 3 x 20 x 4 + 4 x 15 x 3 + 5 x 6 x 2 + 6 x 1 x 1]
= 1/249[36 + 150 + 240 + 180 + 60 + 6]
= 672/249
= 2.7

[2] BINARY 6-CUBE

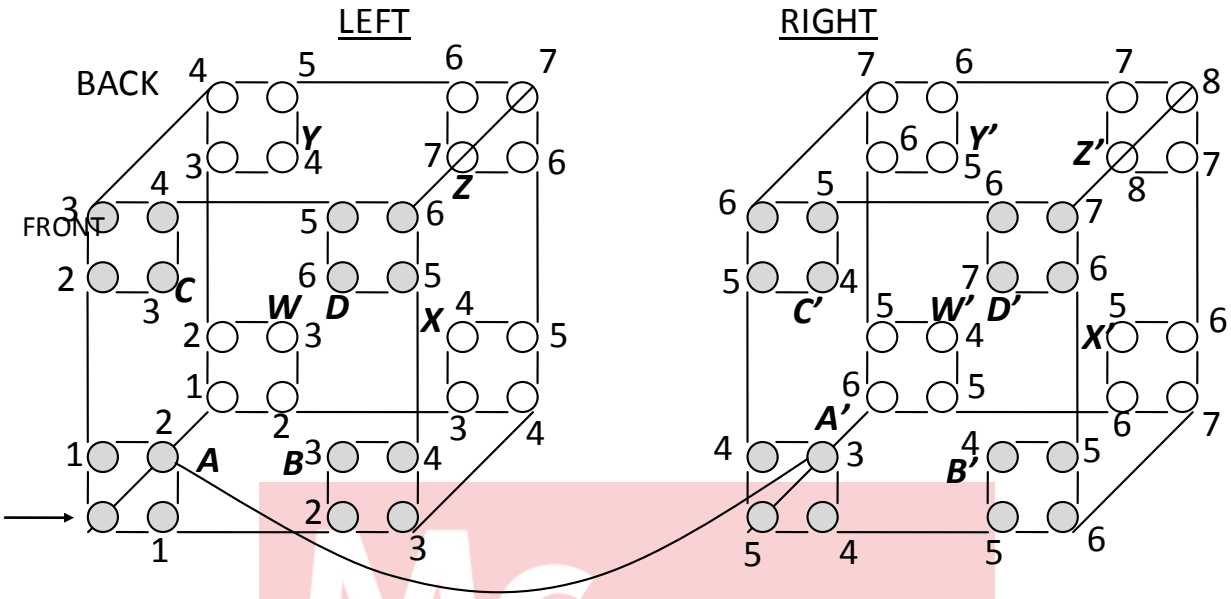
From node [0,0,0,0,0,0] we go to the farthest node [1,1,1,1,1,1] by flipping six indices, in any order. "On the way to the farthest node", by flipping 1, 2, 3, 4 or 5 indices, we can reach all other nodes.

Therefore:
The number of nodes at distance 1 = number of ways to flip 1 index = ${}_6\mathbf{C}_1$.
and so on. Thus,
The number of nodes at distance *k* = number of ways to flip *k* indices = ${}_6\mathbf{C}_k$, *k* = 2.6.

These numbers agree with what was seen above for the 3D TORUS with four nodes to a side. Therefore, the remaining answer for the *binary 6-cube* is the same.

[3] 4D CCC

See the given diagram:
[Compare with Fig. 2.19 (c) in the book.]



Nodes A & A', B & B', C & C', D & D' are in the 'front plane', connected pair-wise. But only the A-A' link is shown above. Nodes W & W', X & X', Y & Y', Z & Z' are in the 'back plane', connected pair-wise (links not shown).

We must now count the number of nodes at various distances from the source node (shown at bottom left, see arrow).

# hops i	1	2	3	4	5	6	7	8
LT # nodes	3	5	7	6	4	4	2	0
RT # nodes	0	0	1	5	9	9	6	2
TOTAL #	3	5	8	11	13	13	8	2
Prob(i)	8/271	7/271	6/271	5/271	4/271	3/271	2/271	1/271

To apply the probability formula, we need to calculate the denominator:
= 3 x 8 + 5 x 7 + 8 x 6 + 11 x 5 + 13 x 4 + 13 x 3 + 8 x 2 + 2 x 1
= 24 + 35 + 48 + 55 + 52 + 39 + 16 + 2 = 271
[This value is used in the last row of the table above.]

The required weighted mean distance is now calculated:
= 1/271[1 x 3 x 8 + 2 x 5 x 7 + 3 x 8 x 6 + 4 x 11 x 5 + 5 x 13 x 4 + 6 x 13 x 3 + 7 x 8 x 2 + 8 x 2 x 1]

= 1/271[24 + 70 + 144 + 220 + 260 + 234 + 112 + 16] = 1080/271 = **3.99**
[Note: This is greater than the value seen in the previous two networks].

12. 8 x 8 ILLIAC mesh, BINARY 6-CUBE, 64 node barrel shifter

It is shown in the book that 8 x 8 ILLIAC mesh is topologically equivalent to a chordal ring of degree 4.

(a) Nodes reachable in **exactly** 3 steps:

Chordal ring of degree 4:

Note: A step along the ring adds +1 or -1 to node index; a step along the chord adds +8 or -8 to node index.

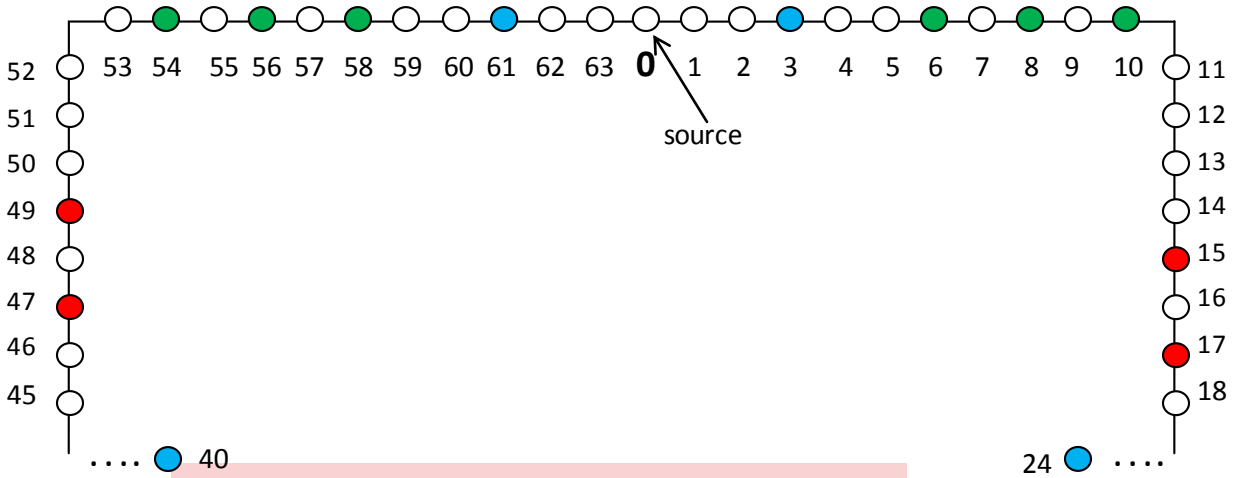
Strictly along the ring, in exactly three steps we can go from N0 to N3, N61. Strictly along the chords, in three steps we can get from N0 to N24, N40 shown in blue below.

After two steps along the ring, we reach N2, N62. From these, in one more chord step, we reach N58, N10, N6, N54 shown in green below.

After one step along the ring, we reach N1 and N63. From these, in two more chord steps, we reach N17, N49, N15, N47 shown in red below.

After one step along the ring, we reach N1 and N63. From these, in one chord step, we reach N9, N57 and N7, N55. From these, in one final ring step, we reach N8, N10, N56, N58, N6, N8, N54, N56 shown in green below.

After two chord steps, we reach N16 and N48. From these, in one ring step, we reach N15, N17, N47, N49 shown in red below.



Thus, total number of nodes reachable in exactly 3 steps = 14.

Binary 6-cube:

Consider that the source node has indices [0,0,0,0,0,0]. Any node reachable in three "hops" will have three of these indices flipped to 1. There are exactly ${}_6C_3$ ways of selecting 3 of the 6 indices to flip. Therefore the number of nodes reachable in three "hops" is ${}_6C_3 = \mathbf{20}$.

64-node barrel shifter:

Note: A step along the ring adds +1 or -1 to node number. A step along a chord adds or subtracts a power of 2 [i.e. 2^k for k = 1,2,3,4,5].

To check reachability in exactly three steps, we need to express the index using addition or subtraction of exactly 3 such terms. First the powers of 2 and then the others.

$1 = 2^2 - 2^1 - 1$
$2 = 2^2 - 1 - 1$

$4 = 2^1 + 1 + 1$
$8 = 2^2 + 2^1 + 2^1$

$16 = 2^3 + 2^2 + 2^2$
$32 = 2^4 + 2^3 + 2^3$

$3 = 1 + 1 + 1$
$5 = 2^1 + 2^1 + 1$
$6 = 2^2 + 1 + 1$
$7 = 2^2 + 2^1 + 1$
$9 = 2^2 + 2^2 + 1$

$10 = 2^3 + 1 + 1$
$11 = 2^3 + 2^1 + 1$
$12 = 2^3 + 2^1 + 2^1$
$13 = 2^3 + 2^2 + 1$
$14 = 2^3 + 2^2 + 2^1$

$15 = 2^3 + 2^3 - 1$
$17 = 2^3 + 2^3 + 1$
$18 = 2^3 + 2^3 + 2^1$
$19 = 2^4 + 2^2 - 1$
$20 = 2^3 + 2^1 + 2^1$

$21 = 2^4 + 2^2 + 1$
$22 = 2^4 + 2^2 + 2^1$

$26 = 2^4 + 2^3 + 2^1$
$27 = 2^5 - 2^2 - 1$

$23 = 2^4 + 2^3 - 1$
$24 = 2^4 + 2^2 + 2^2$
$25 = 2^4 + 2^3 + 1$
$28 = 2^5 - 2^1 - 2^1$
$29 = 2^5 - 2^1 - 1$
$30 = 2^5 - 1 - 1$
$31 = 2^5 + 1 - 2^1$

Similarly, by changing + and - signs, we can go from N0 to all nodes N33 - N63.
In conclusion, in exactly 3 steps, **we can reach all the nodes of the barrel shifter**.
This verifies that the diameter of this barrel shifter is 3.

(b) The given wording seems to refer simply to the network diameter. It is shown in the text that:

(i) for *nxn* ILLIAC mesh the diameter is *n*-1 (page 76),

(ii) for the binary 6-cube, the diameter is 6, and

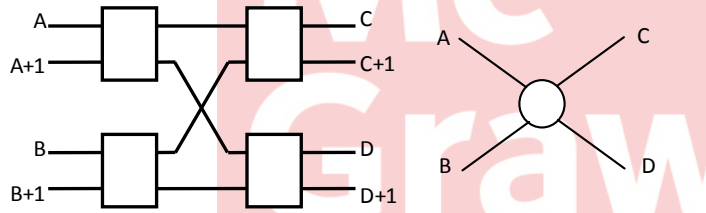
(iii) for the barrel shifter of 2^k nodes the diameter is k/2 (on page 70).

Accordingly, the answer is: 7, 6 and 3, respectively.

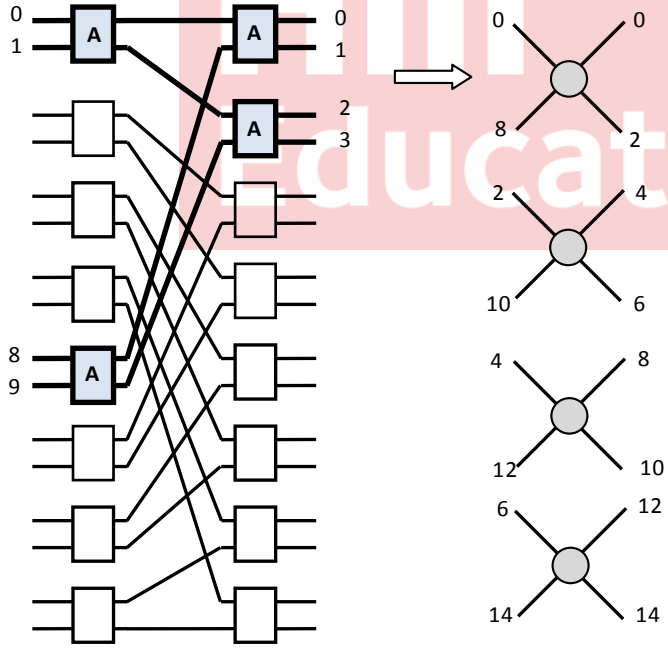
(c) Since 1024 = 32x32 = 2^{10}, the answer is: 31, 10 and 5, respectively.

15. Showing the topological equivalence of Omega, Baseline and Flip networks.

Consider the interconnection of four 2 x 2 switch modules as shown on the left below. Without any loss of information, that network of switches can be depicted in graph form as shown on the right.



On the right, the edge marked A represents the two input lines A, A + 1 as well as the switch module connected to them. And similarly for edges B, C and D.
The central node represents the interconnection between inputs and outputs.



With this representation, one stage of the Omega network (on the left) is represented as shown on the right. It appears simpler only because we have not written the left edge numbers in sequential order.

Now, we can see that any single stage of any MIN is simply the graph shown on the right. ONLY THE EDGE NUMBERS ARE DIFFERENTLY PERMUTED!

This is the classic definition of graph isomorphism. Thus parts (a), (b) and (c) of the problem are solved easily.

Chapter 3 Selected Solutions

[General note: There has not been any opportunity for interaction between the two co-authors, to achieve needed clarifications. This is the reason behind some of the notes written hereunder.]

1. Number of instructions in each of the four parallel parts = $2 \times 10^6/4 = 5 \times 10^5$
Overhead instructions in each part = 50000
→ Overhead fraction= $5 \times 10^4/5 \times 10^5 = 0.1$ [→equivalent to 10%]
CPI for memory reference (revised value) = 12

(a) Revised CPI = $0.6 \times 1 + 0.18 \times 2 + 0.12 \times 4 + 0.1 \times 12 = 2.64$

(b) MIPS rating should increase by a factor of 4, but due to 10% overhead, it will increase by a factor $4 \times 10/11 = 3.64$

Therefore, MIPS rating = $3.64 \times f/(\text{CPI} \times 10^6) = 3.64 \times 400/2.64 = 551.5$

[For a fair comparison, speed-up achieved on the 4 processor system should be reduced in proportion to the overhead. This gives effective MIPS rating.]

(c) 3.64, as seen above

(d) For every 10 units of effective work, there is one unit of overhead (10%).
Therefore efficiency = $(10/11) \times 100 = 90.91\%$.

2. Let S and V be fraction of program running in scalar and vector modes, resp.ectively.
Thus, S + V = 1.

Vector mode is 9 times more efficient. Let M be the MIPS rating of the scalar mode; therefore, the MIPS rating of vector mode is 9M.

Computation performed = $(S \times M + V \times 9M) \times T$, where T is the time taken.
For the given program, we have: $V = 0.25$, $S = 0.75$

→ Computation performed = $(0.75M + 2.25M)T = 3MT$
[Out of this, $0.75MT$ is in scalar mode, and $2.25MT$ is in vector mode.]

(a) If the entire computation is performed in scalar mode, MIPS rating = M.
→Time taken in scalar mode = computation / MIPS rating = $3MT/M = 3T$.

Therefore, the effective speedup under given condition = $3T/T = 3$.

Percentage of **computation** that is vectorized = (2.25/3) x 100 = 75%

[Note: IMHO, the percentage of **code** that is vectorized cannot be determined.]

(b) With doubled performance: vector mode MIPS = 18M

→ Time spent in vector mode = 2.25MT/18M = 0.125T
Time spent in scalar mode = 0.75T as before
→ Total time spent = 0.875T
→ Speedup = 3T/0.875T = 3.43 [compared with previous speedup of 3]

(c) Percentage of computation that is vectorized remains the same, i.e. 75%. This will not change with the compiler. [See note above.]

3. Assume that computation time is proportional to program code.

[See note in previous exercise. A 100 line loop in a program can be executed 10 times or 10^6 times. So clearly computation performed is NOT proportional to code length. Therefore solving such exercises in terms of "percentage of code" is not possible, nor instructive. In fact this difference should be made clear to students.]

Let the total computation performed be C, in terms of number of instructions.

(a) Fraction of computation that is parallelized over *n* processors = α (given).
→ Time taken for the parallelized part = $\alpha C/(nx)$
Time taken for the sequential part = $(1 - \alpha)C/x$
→ Total time = $\alpha C/(nx) + (1 - \alpha)C/x$
→ Effective MIPS
= computation performed / total time
= $C/[\ \alpha C/(nx) + (1 - \alpha)C/x\ ]$
= $x/[\ \alpha/n + (1 - \alpha)\ ]$

(b) Substitute the given numerical values.
→ $4000 = 400/[\alpha/16 + (1 - \alpha)]$
→ $\alpha/16 + (1 - \alpha) = 0.1$
→ $15/16\alpha = 0.9$
→ $\alpha = 0.96$
[96% of the computation must be paralleized over 16 processors]

4. Apparent oversight: Performance ratio between regular and enhanced modes has **not** been specified. Let this ratio be R; i.e. regular mode MIPS = m, enhanced mode MIPS = mR. Let the total computation be C.

For program running only in regular mode, time taken = C/m [1]

With mix of regular & enhanced modes, for a given value of α:
Time spent in regular mode = αC/m
Time spent in enhanced mode = (1 - α)C/(mR)
Total time spent = αC/m + (1-α)C/(mR) [2]

Speedup (with reference to regular mode)
= [1] divided by [2]
= $1/[\alpha + (1-\alpha)/R]$
= $R/[(R-1)\alpha + 1]$
→ Let this expression be denoted by $f(\alpha)$.
→ Required harmonic mean = $\int_a^b f(\alpha)d\alpha / [b-a]$

This evaluates to $[R/\{(b-a)(R-1)\}] \times \ln[\{b(R-1)+1\} / \{a(R-1)+1\}]$

With a = 0 and b = 0, we get: Required harmonic mean = $[R/(R - 1)] \times \ln(R)$

5. (a) This is essentially the same derivation as given on pp. 92–93 of the text-book. The only difference in problem definition here is that — instead of different programs 1, 2, 3, 4 — we have now different execution modes of a program mix.

So we can use expression 3.12 on p. 93 to get the weighted harmonic mean execution rate, and the inverse of that for the weighted mean execution time per instruction. Accordingly:

(b) $R_h^* = [f_1/R_1 + f_2/R_2 + f_3/R_3 + f_4/R_4]^{-1}$
$= [0.4/400 + 0.3/800 + 0.2/1100 + 0.1/1500]^{-1}$
$= [0.001 + 0.000375 + 0.000181 + 0.000067]^{-1}$
$= [1623 \times 10^{-6}]^{-1}$
= 616 MIPS

The inverse of this is the weighted mean execution time per instruction.

Reason why system MIPS rating is not proportional to the number of processors: Communication overheads, contention for memory access, some additional work to be done by the OS, etc. [as explained at a few points in the textbook].

(c) With the new weights:
$R_h^* = [f_1/R_1 + f_2/R_2 + f_3/R_3 + f_4/R_4]^{-1}$
$= [0.1/400 + 0.2/800 + 0.3/1100 + 0.4/1500]^{-1}$
$= [0.00025 + 0.00025 + 0.000273 + 0.000267]^{-1}$
$= [1040 \times 10^{-6}]^{-1}$
= 962 MIPS

The inverse of this is the weighted mean execution time per instruction.

6. Amdahl's law: Parallel program runs either in sequential mode (one processor), or symmetrically on all N processors. As a result, with large N, system performance is limited by the 'sequential bottleneck'.

Gustafson's law: Problem can be scaled to make maximum possible use of available N processors, e.g. by increasing the number of grid points in a finite element method. As a result, more accurate result can be obtained in the given time, and the problem of 'sequential bottleneck' is reduced in that sense.

Sun & Ni's law: This law explicitly considers the effect of aggregate memory on the scaled problem size. If the problem size scales faster than linearly with aggregate memory (N x M), then the speedup achievable is higher than that under the previous two laws above.

9. Table of Execution times (in seconds) — from Problem 1.6 (repeated for convenience):

	Computer A	Computer B	Computer C
Program 1	1	10	20
Program 2	1000	100	20
Program 3	500	1000	50
Program 4	100	800	100

(a) Total execution time (in seconds) for program mix:

	Computer A	Computer B	Computer C
Total ex. time	1601	1910	190

Number of instructions executed on each computer = 4×10^9.
→ Mean execution time per instruction = (total execution time)/(4×10^9).

This is given in the following table:
Mean execution time per instruction (in 10^{-6} seconds):

	Computer A	Computer B	Computer C
Mean ex. time per inst.	0.40025	0.4775	0.0475

(b) Harmonic mean MIPS rate of each machine is the inverse of part (a) answer, and given in the table:

	Computer A	Computer B	Computer C
Har. mean MIPS rate	2.50	2.09	21.05

(c) The ranking is seen above.
The harmonic mean MIPS rate calculated here agree well with the conclusions drawn in the solution to Problem 1.6.

3.10 (a) Given fully on p. 114 (top half), along with the assumptions under which the derivation is valid.

(b) Substitute G(n) = 1, as shown later on p. 114.

(c) Substitute G(n) = n, as shown later on p. 114.

(d) First consider the ratio S*/S':

$S^* = [W_1 + G(n)W_n] / [W_1 + G(n)W_n/n]$
$S' = [W_1 + nW_n] / [W_1 + W_n]$

$\rightarrow S^*/S' = [W_1^2 + \{G(n)+1\}W_1W_n + G(n)W_n^2] / [W_1^2 + \{G(n)/n+n\}W_1W_n + G(n)W_n^2]$

First and third terms in numerator and denominator are equal.
Subtract the middle terms in numerator and denominator:
$\rightarrow G(n) +1 - [G(n)/n + n] = (n - 1)G(n)/n - (n -1)$

This is > 0 when G(n) > n $\rightarrow$ numerator > denominator $\rightarrow S^* > S'$

Now consider the ratio S'/S:

$S' = [W_1 + nW_n] / [W_1 + W_n]$
$S = [W_1 + W_n] / [W_1 + W_n/n]$

Ratio = $[W_1^2 + W_n^2 + (n+1/n)W_1W_n] / [W_1^2 + W_n^2 + 2W_1W_n]$
Once again first and third terms in numerator and denominator are equal.
Therefore, this ratio > 1 if n > 1 $\rightarrow S' > S$

11. (a) That 1/n <= E(n) <= 1 is already shown on p. 95.

To show that the same limits apply to U(n), consider 3.18: U(n) = O(n)/[nT(n)].
Now note that O(n) <= nT(n) because O(n) <= nT(1) and T(n) <= T(1).
Therefore since numerator <= denominator, we have U(n) <= 1.

To show that E(n) <= U(n), compare 3.16 and 3.18, and note that O(n) >= O(1) = T(1).

(b) The limits on 1/E(n) follow from what is already shown in the book on p. 95.

The limits on R(n) follow from the fact that O(n) <= nO(1).

To show that R(n) <= 1/E(n), compare 3.16 and 3.17, note that O(1) = T(1), and that O(n) <= nT(n) since at most n processors are used at any time.

(c) Follows straightway by substituting the definitions on the right side.

(d) Relationships of (a) are visible in fig. 3.4. Relationships of (b) need a little extra work of calculating 1/E(n) for various values of n. Straightforward.

13. (a) For 100 x 10^6 operations, MIPS ratings are calculated as in the first two rows:
[Note: $\mathbf{R_h}$ is calculated from total time for 200 x 10^6 ops.]

	Computer A	Computer B	Computer C
Problem 1	100	10	5
Problem 2	0.1	1	5
$\mathbf{R_a}$	50.05	5.5	5
$\mathbf{R_h}$	0.2	1.82	5

→ When the two performances differ a great deal, $\mathbf{R_h}$ is a better measure than $\mathbf{R_a}$.
The difference between $\mathbf{R_a}$ and $\mathbf{R_h}$ can be understood as follows:
Suppose a car is driven at speed X for half of the distance, and speed 2X for the remaining half of the distance. The average speed is NOT 1.5X. It is actually:

D/[(D/2)/X + (D/2)/(2X)] = 1/[0.5/X + 0.25/X] = X/0.75 = 1.33X

(b) The problem now seems to be restated as follows:

Benchmark 1 now runs at 10 MIPS on all three computers. Therefore, its running time is 100/10 = 10 seconds on all three computers. The weights are also different.

Therefore the revised problem statement now (running times in seconds) is:

	Computer A	Computer B	Computer C
Problem 1 (f_1=0.8)	10	10	10
Problem 2 (f_2=0.2)	1000	100	20

$\mathbf{R_a}$ and $\mathbf{R_h}$ are now calculated as follows:

	Computer A	Computer B	Computer C
$\mathbf{R_a}$	0.8x10 + 0.2x0.1 = **8.02**	0.8x10 + 0.2x1 = **8.2**	0.8x10 + 0.2x5 = **9**
$\mathbf{R_h}$	200/[0.8x10 + 0.2x1000] = **0.962**	200/[0.8x10 + 0.2x100] = **7.14**	200/[0.8x10 + 0.2x20] = **16.67**

→ Once again, this shows that $\mathbf{R_h}$ is the more meaningful measure.

(c) This is similar to part (b), and the final table is shown below:

	Computer A	Computer B	Computer C
$\mathbf{R_a}$	0.2 x 10 + 0.8 x 0.1 = **2.08**	0.2 x 10 + 0.8 x 1 = **2.8**	0.2 x 10 + 0.8 x 5 = **6**
$\mathbf{R_h}$	200/[0.2 x 10 + 0.8 x 1000] = **0.249**	200/[0.2 x 10 + 0.8 x 100] = **2.44**	200/[0.2 x 10 + 0.8 x 20] = **11.67**

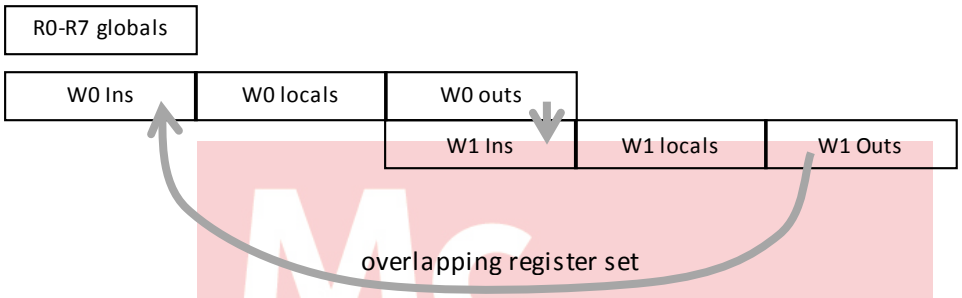
→ Differences between $\mathbf{R_a}$ and $\mathbf{R_h}$ have narrowed now, because more weight is now given to Problem 2, which has very different performance on the three machines.

(d) The conclusion from these calculations is that [based on $\mathbf{R_h}$ - for reasons given], the overall performance improves as we go from computer A to computer B, and again as we go from computer B to computer C.

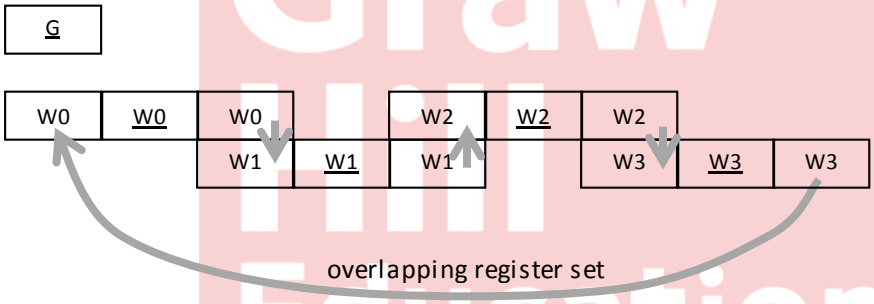
CHAPTER 4 Selected Solutions

Note: There are only a few "solvable" exercises given in this chapter. The rest require simply that appropriate points be studied in the chapter and described in student's own words.

4.8 (a) As seen in figure. Since overlapping set of registers counts only once, the total number of registers is: 8 globals + 4 x 8 = 40.



(b) Schematic arrangement will be:



Each arrow indicates an overlapping **IN-OUT** set of registers.
Letter G indicates globals. W1, W2, W3 and W4 indicate respective locals.

Total number of registers = 8 + 8x8 = 72.

4.11 (a) Average cost of the entire memory
= total cost / usable memory size
= $[M_1c_1 + M_2c_2] / M_2$ assuming memory sizes in kilobytes
= $c_2 + M_1c_1/M_2$
This value approaches c_2 only when $M_1 << M_2$.

(b) Hit ratio to M_1 is *h*.
Therefore, (1- h) fraction of requests are served by M_2.

Therefore, effective access time
= average or mean access time
= $h * t_1 + (1 - h) * t_2$

(c) $E = t_1/t_a$
$= t_1/[h * t_1 + (1 - h) * t_2]$
Divide numerator and denominator by t_1, giving:
$E = 1/[h + (1 - h)*r]$

(d) To be plotted.

(e) $0.95 < 1/[h + (1 - h)*100]$
This gives:
$100 - 99*h < 1.05$, which means $h > 98.95/99$!
This is indeed a very high hit ratio!

Parts (d) & (e) of the exercise point to the need for multilevel cache memories.

4.12 See table below for parts (a), (b) and (c).

	Hit ratio, *h*	Size (kB)	Average access time (ns) = $20h + 200(1-h)$	Total cost ($) = 4096 x 0.2 + cache size x 4
Cache 1	0.7	64	14 + 60 = 74 ns	819.2 + 256 = 1075.2
Cache 2	0.9	128	18 + 20 = 38 ns	819.2 + 512 = 1331.2
Cache 3	0.98	256	19.6 + 4 = 23.6 ns	819.2 +1024 = 1843.2

Note: Average cost per byte is proportional to total cost, since M_2 is fixed at 4 MB.

Ranking is as expected. With larger cache, access time improves but cost goes up. Product of cost and average access time gets better as we go from smaller to larger cache.

4.15 (a) The first row in the table below shows referenced virtual (i.e. logical) page number.
The second row shows whether a page fault occurs ('F').
The last four rows shows the virtual page numbers occupying available (4) page frames (in LRU order, with faulting page number just below the letter 'F').
Each time, by inspection, the LRU page is removed (shaded in previous column).

1	0	2	2	1	7	6	7	0	1	2	0	3	0	4	5	1	5	2	4	5	6	7	6	7	2	4	2	7	3	3	2	3
F	F	F			F	F		F		F		F		F	F	F		F			F	F			F	F			F			
1	0	2	2	1	7	6	7	0	1	2	0	3	0	4	5	1	1	2	4	5	6	7	6	7	2	4	2	7	3	3	2	3
	1	0	0	2	1	7	6	7	0	1	2	0	3	0	4	5	5	5	2	4	5	6	7	6	7	2	4	2	7	7	3	2
		1	1	0	2	1	1	6	7	0	1	2	2	3	0	4	4	1	5	2	4	5	5	5	6	7	7	4	2	2	7	7
					0	2	2	1	6	7	7	1	1	2	3	0	0	4	1	1	2	4	4	4	5	6	6	6	4	4	4	4

Total number of page faults = 17 (counted from second row).
Therefore, main memory hit ratio = (33 - 17)/33 = 16/33.
However, IMHO, the first four page faults should not count, because initially they must be brought into main memory.
Therefore, hit ratio = [33 - (17 - 4)]/33 = 20/33.

(b) See the given table.

Except for the change in page replacement policy, the meaning of various rows is the same. The last four entries in any column are in FIFO order, with faulting page number below 'F'.

1	0	2	2	1	7	6	7	0	1	2	0	3	0	4	5	1	5	2	4	5	6	7	6	7	2	4	2	7	3	3	2	3
F	F	F			F	F			F		F	F		F	F	F		F			F	F				F			F		F	
1	0	2	2	2	7	6	6	6	1	1	0	3	3	4	5	1	1	2	2	2	6	7	7	7	7	4	4	4	3	3	2	2
	1	0	0	0	2	7	7	7	6	6	1	0	0	3	4	5	5	1	1	1	2	6	6	6	6	7	7	7	4	4	3	3
		1	1	1	0	2	2	2	7	7	6	1	1	0	3	4	4	5	5	5	1	2	2	2	2	6	6	6	7	7	4	4
					1	0	0	0	2	2	7	6	6	1	0	3	3	4	4	4	5	1	1	1	1	2	2	2	6	6	7	7

Number of page faults = 17.
In this instance, the answer happens to be the same as in part (a).

(c) In this instance, (circular) FIFO happens to match the performance of LRU. The idea seems to be to show that FIFO is a good approximation to LRU (assuming a random page reference pattern). Also please note (ref. p 175): There does not seem to be any significant difference between FIFO and Circular FIFO.

4.17 (a) Effective access time $t_{eff} = h * t_1 + (1 - h) * t_2$, where h is the cache hit ratio.

(b) Total memory cost = $c_1 s_1 + c_2 s_2$ [cost and size both in bytes]

(c) (i) We must solve the equation:

$15000 = 512 * 1024 * 0.01 + s_2 * 0.0005$

Giving s2 = 19514240 bytes (slightly less than 20 Mbytes).

(ii) We must solve the equation:

$40 = 0.95 * 20 + 0.05 * t_2$

Giving t_2 = 420 ns.

CHAPTER 5 Selected Solutions

5.1 96 signal lines are available on the bus. These have to be shared between data, address and various control signals (see fig. 5.1). This division is, however, not specified.

Processor word size = 64 bits and memory address = 40 bits (since memory size is 2^{40} words). We make the following assumptions:

(1) Either 64 data bits or 40 address bits can be placed on the bus in one cycle, but not both.

(2) In one bus cycle of memory access, the memory address and the required control signals are sent on the bus to the target memory module number.

(3) In four subsequent cycles, four memory words of 64 bits each (i.e. total 32 bytes) are transferred (i.e. read or written).

(4) Important: With the use of cache buffers on memory modules, the 100 ns memory latency is not allowed to slow down bus operations.

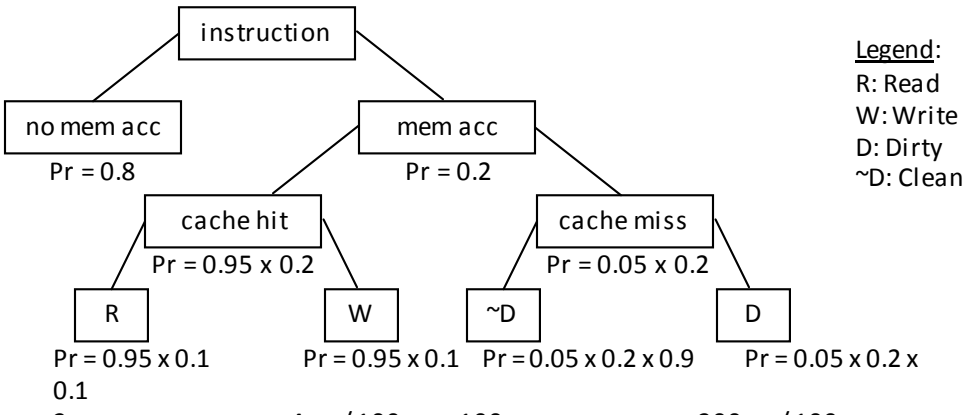
With these assumptions, the specified four words per memory access are achieved in total five cycles of the bus. Now the question can be answered:

(a) 200 MHz → 5 ns cycle. Therefore, the theoretical maximum bus bandwidth = 64 bits per 5 ns = 12.8 GHz.

(b) Effective processor-memory bandwidth = 32 bytes in 5 bus cycles = 256 bits per 25 ns = 10.24 GHz.
But effective bandwidth available to one processor = 10.24/4 = 2.56 GHz.

(c), (d) and (e): Please see Section 5.1 on Bus Systems.

5.4 (a) The following tree diagram sets out the given data (for one instruction):



In the diagram, each event is associated with a probability (= 1 for root node), based on the data provided. Probability of parent node = sum of probabilities of child nodes.

Leaf nodes also have time of execution associated with them, as specified. Where two times of execution are given (4 ns/100 ns and 200 ns/100 ns), the first figure applies to write-back cache, and the second figure applies to write-through scheme.

The 200 ns figure above includes 100 ns to write back a dirty cache block, and another 100 ns to read the required cache block (which caused the cache miss).

Effective memory access time for write-back scheme
= probability-weighted average of write-back timings
= 0.95 x 0.1 x 2 + 0.95 x 0.1 x 4 + 0.05 x 0.2 x 0.9 x 100 + 0.05 x 0.2 x 0.1 x 200
= 0.19 + 0.38 + 0.9 + 0.2
= 1.67 ns

Effective memory access time for write-through scheme
= probability - weighted average of write-through timings
= 0.95 x 0.1 x 2 + 0.95 x 0.1 x 100 + 0.05 x 0.2 x 0.9 x 100 + 0.05 x 0.2 x 0.1 x 100
= 0.19 + 9.5 + 0.9 + 0.1
= 10.69 ns

The above figures represent average memory access delay per instruction. With this data, write-back scheme performs better. Write-through suffers from the 100 ns needed on each write operation (which is 10% of all instructions).

(b) Given: Processor MIPS rate = 500 assuming 100% cache hit. Therefore, we need to calculate the effect of cache misses on the MIPS rate. We assume write-back cache, because the effective memory access time for write-through is very bad.

With 100% cache hit rate, average cache access time per instruction
= 0.95 x 0.1 x 2 + 0.95 x 0.1 x 4 = 0.57 ns [half read, half write operations]

Thus, the processor completes one instruction in 2 ns (i.e. 500 MIPS) after we include (that is, factor in) this 0.57 ns average cache access time.

With 95% cache hits, the average memory access time is 1.67 ns, i.e. it increases by 1.1 ns (the last two terms in the sum of four terms seen above).

Thus, we can assume that the average instruction execution time increases to 2 + 1.1 = 3.1 ns. Therefore, the reduced MIPS rate = 1000/3.1 = 322.6 MIPS.

With 16 processors, the absolute theoretical upper MIPS bound is therefore 322.6 x 16 ~ = 5160. This assumes calculation no loss of cycles in bus contention.
In actual fact, each processor gets a cache miss with a probability of 0.05. Therefore with 16 processors (since 16x0.05 = 0.8), there will be plenty of bus contention. If a processor does not get bus access, it must wait at least another 100 ns!

Therefore the theoretical MIPS upper bound of 5160 cannot be achieved, and in fact cannot even be approached! The idea behind part (b) seems to be to show up the performance limiting effect of bus-based multiprocessor systems.

5.7 (a) Attached MATLAB program Problem 5.7 simulates the 90 execution orders and shows each output sequence produced. To see why the count is 90, consider the following:

There are 6 program statements: a, b, c, d, e, f and g. We know that 6! = 720.

However, statement 'a' must precede statement 'b' (program order).
Therefore with this constraint, half the statements where 'b' precedes 'a' must be dropped. Thus the number of statements reduces to 720/2 = 360.

Apply the same argument with 'c' and 'd', and the number drops to 360/2 = 180.
Now again apply the argument with 'e' and 'f', and the number drops to 180/2 = 90.

(b) NO. All possible 6-tuples of '0' and '1' cannot be generated even when we consider all 720 permutations of the six statements. This is because a sequence like '111000' cannot be generated. Reason: Once the value '1' is printed for the value of A, B or C, then it cannot be followed by '0' for the same variable, since none of the six statements changes any value from '1' to '0'.

(c) '011001' is not a valid output pattern for the following reason:
When we scan the output string from right to left [i.e. from the last output back towards the first], at no stage can the number of '0's exceed the number of '1's.
Why so? Because, in program order, every '0' eventually becomes '1'.

In the output string '011001', when we scan three characters from the right end, we see '001', which has two '0's and a '1'. Since both the '0's must become '1' by the time the last statement is executed, this string is not a possible output if all the statements are executed in program order.

(d) With non-atomic memory access, an invalid copy of a variable can in theory remain in a processor cache, from which it is sent to output. Thus we can see a '0' in output even when the memory variable has meanwhile been changed to '1' by another processor. [This assumes that the variable is cached in the first processor prior to the program sequences shown]. So '011001' is a possible output.

5.8 Here $m = 8$, $n = 64$. Thus $r = 3$ and $s = 6$. Block size $b = 8$ therefore $w = 3$.
Memory address size = $s + w = 9$ bits.

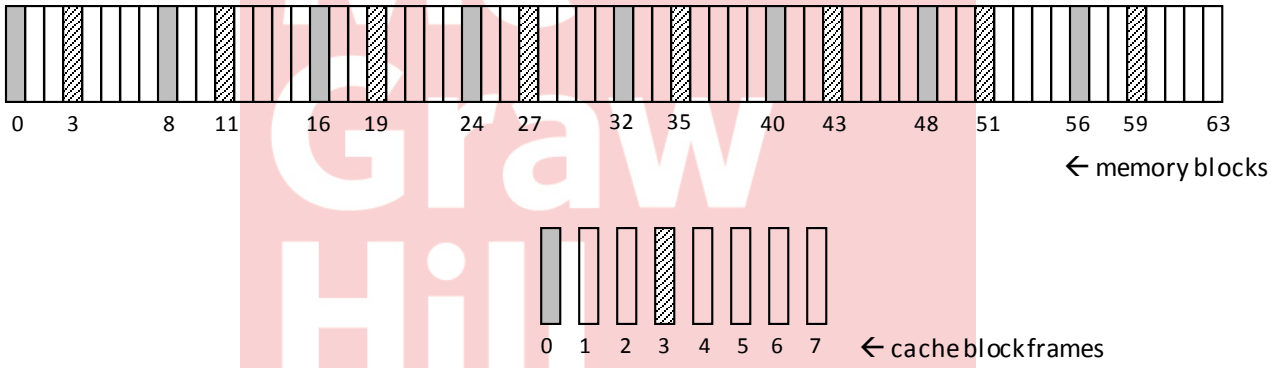
(a) Direct mapping:

Memory block number = j maps to cache block frame number = j (modulo 8)

It is difficult to show all 64 lines, but the two shading patterns used below correspond to: j (modulo 8) = 0 and j (modulo 8) = 3 respectively.

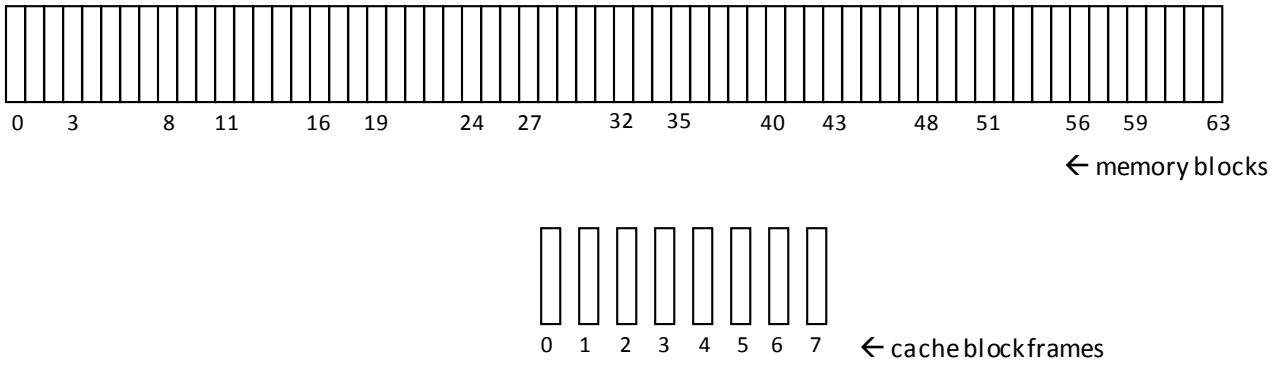
Lines should be imagined from memory blocks to similarly shaded cache block frames (and then yet more lines between the remaining memory blocks and cache frames)!

Tag field size = $s - r = 3$ bits. Tag fields are not shown in the figure.
Memory address = 3 bits tag + 3 bits block number + 3 bits word number



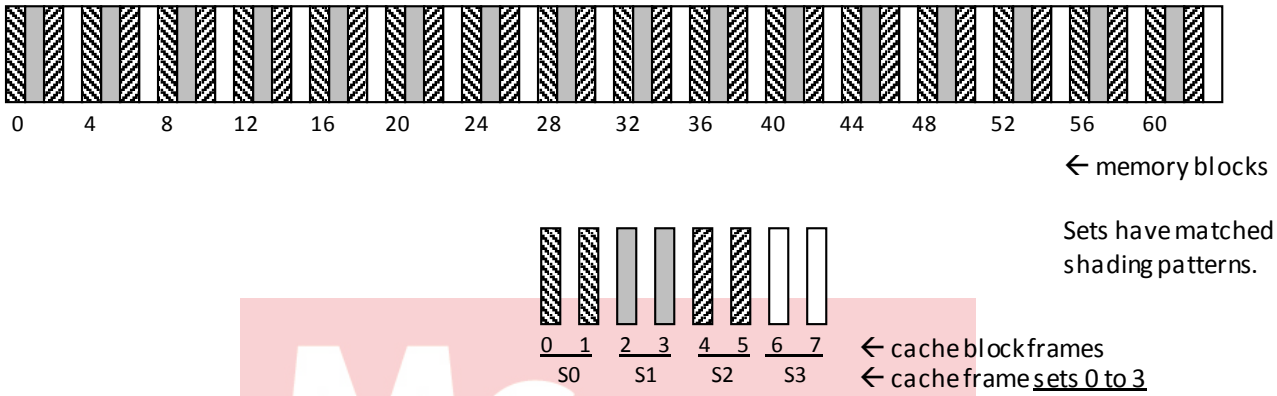
(b) Fully associative mapping:

Lines go from every memory block to every cache block frame. Too many to show!
Tag field size = $s = 6$ bits.
Memory address = 6 bits tag + 3 bits word number



2-way set associative mapping:

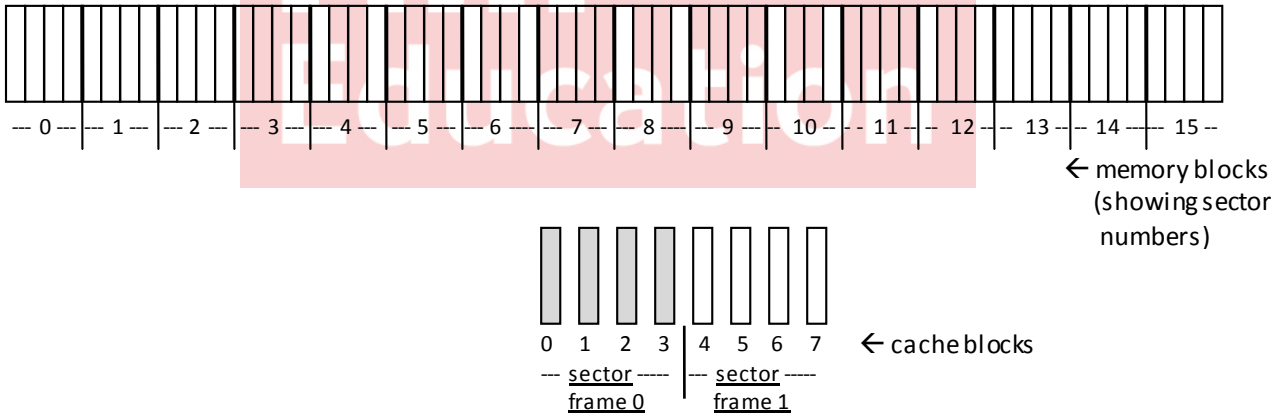
Using shading patterns, the four sets of cache blocks is shown below.
Since we have 2-way set associative cache, each set has two blocks.



For 2-way set associative cache, $v = 2^d = 8/2 = 4$ and therefore $d = 2$ (set number).
Therefore tag field size = $s - d = 6 - 2 = 4$ (to identify one block out of 16).
Memory address = 4 bit tag + 2 bit set number + 3 bit word number

(c) Sector-mapped cache with 4 blocks per sector:

See diagram below.
Memory address = 4 bit tag + 2 bit block number + 3 bit word number



5.9 (a) From the given data:

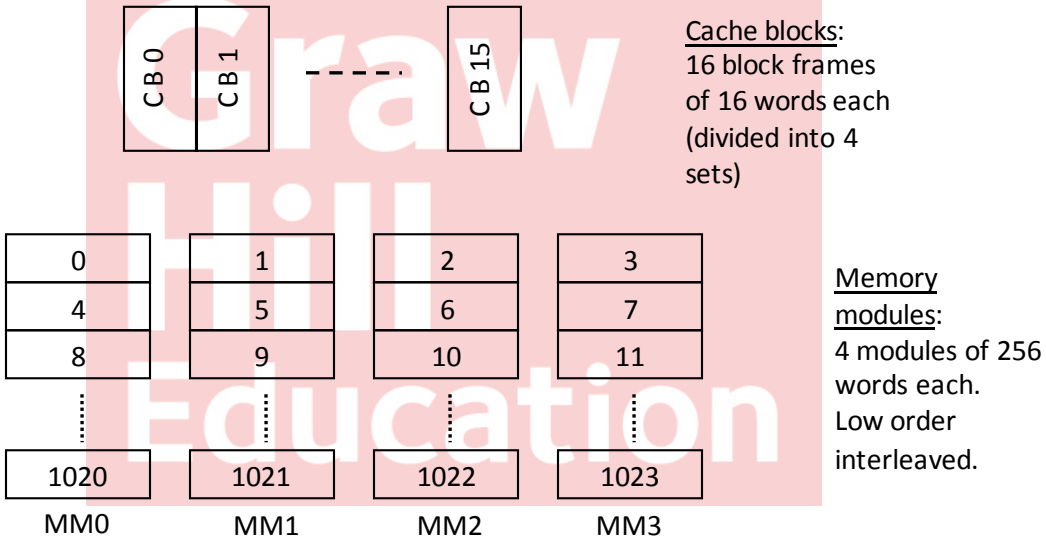
Main memory = 4M words.
Therefore address size = 22 bits (since 2^{22} = 4M).

Cache block = 8 words. So word number field, $w = 3$ bits.
Number of blocks in main memory = 4M/8 = $2^{22}/2^3 = 2^{19}$. Thus $s = 19$.
Set size = 256 words = 32 cache blocks (since 32x8 = 256).
Therefore number of sets in cache = 64K/256 = 256 = $v = 2^d$.
So set number $d = 8$ bits.
Tag bits needed = $s - d = 19 - 8 = 11$.

The following diagram summarizes this information. Based on these numbers, if required, a diagram similar to Fig. 5.11 (a) can be drawn.

(b) $t_{eff} = 0.95x5 + 0.05x40 = 4.75 + 2 = 6.75$ ns.

5.10 (a) Main memory modules and cache as specified are shown below:

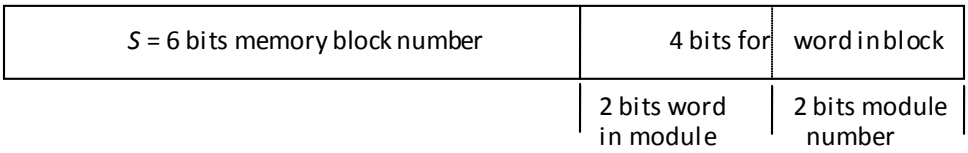


(b) Number of blocks in main memory = 1024/16 = 64.
Number of block frames in cache = 256/16 = 16.

(c) Total memory address size = $\log_2 1024 = 10$ bits.
Lowest order 2 bits specify memory module. Higher order 8 bits specify offset of word within memory module. Because of low-order interleaving, each 16 word cache block is distributed into the four memory blocks.

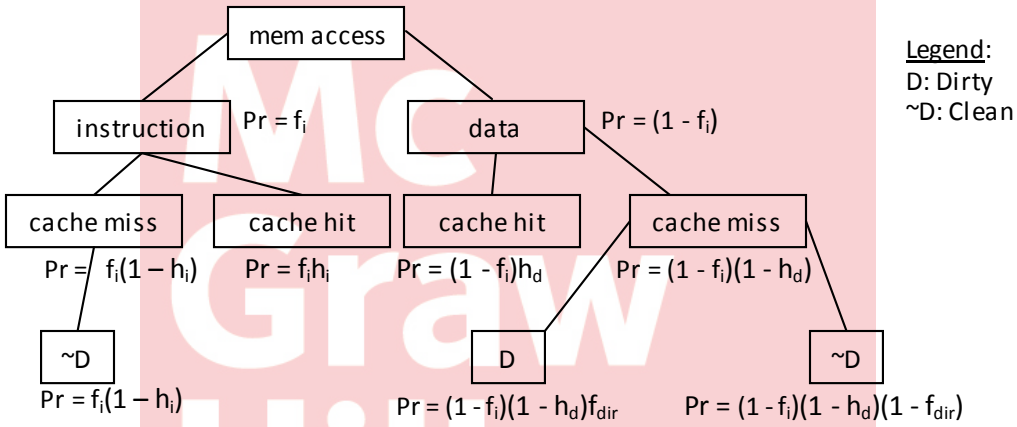
(d) Cache frames are divided into four sets; so $v = 2^d = 4$ and set number $d = 2$.
[Thus each set contains 4 cache frame blocks.]

Thus the 10 bit memory address gets divided into the following fields:



From this, we see $s = 6$. We already have $d = 2$.
So tag field size = $s - d = 4$ bits.
With these numbers, the cache design is as in Fig. 5.11 (a). / HR = h_d

5.11 (a) Consider the probability diagram below:



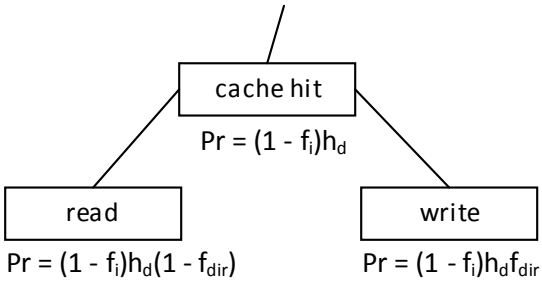
We are also given that cache access time = c clock cycles, and memory access time = b clock cycles. As before (Prob. 5-4) we assume that replacing a dirty cache block takes two memory cycles (1 write + 1 read).

Five relevant events are: (1) i-cache miss, (2) i-cache hit, (3) d-cache hit, (4) d-cache miss on dirty block, and (5) d-cache miss on clean block.

These five events are seen as leaf nodes in the above diagram.

Therefore expected (effective) memory access time
= probability weighted average of the above four access times
= $b*f_i(1-h_i) + c*f_i h_i + c*(1-f_i)h_d + 2b*(1-f_i)(1-h_d)f_{dir} + b*(1-f_i)(1-h_d)(1-f_{dir})$

(b) We assume that the fraction of data writes = f_{dir}. Note that only data writes will invalidate a cache block, not data read. So now the data cache hit leaf node in the above diagram changes as shown below:



On a data write operation with cache hit: (i) the local cache will be written, and (ii) the remote caches (of other processors) will be invalidated. These two hardware operations can proceed in parallel. The first takes 'c' clock cycles, while the second takes 'i' clock cycles. Therefore, in parallel, the time taken will be max(c,i).

Therefore the time taken for the above two leaf node operations is, respectively, 'c' clock cycles and max(c,i) clock cycles. The expression for effective access time is therefore now given by:

$$t_{eff} = b*f_i(1-h_i) + c*f_i h_i + c*(1-f_i)h_d(1 - f_{dir}) + \max(c,i)*(1-f_i)h_d f_{dir}$$
$$+2b*(1-f_i)(1-h_d)f_{dir} + b*(1-f_i)(1-h_d)(1-f_{dir})$$

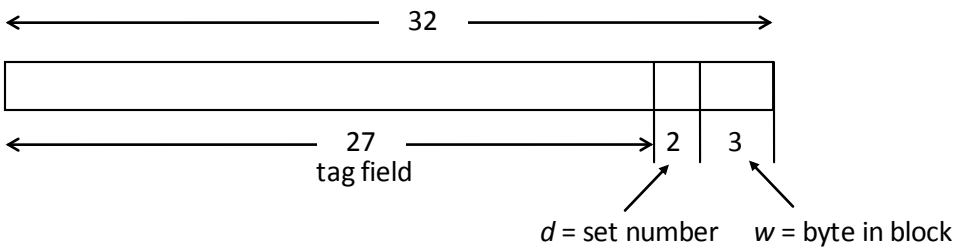
Note:

(1) Bus contention amongst the processors is not considered here.

(2) Terms not highlighted remain the same as in part (a).

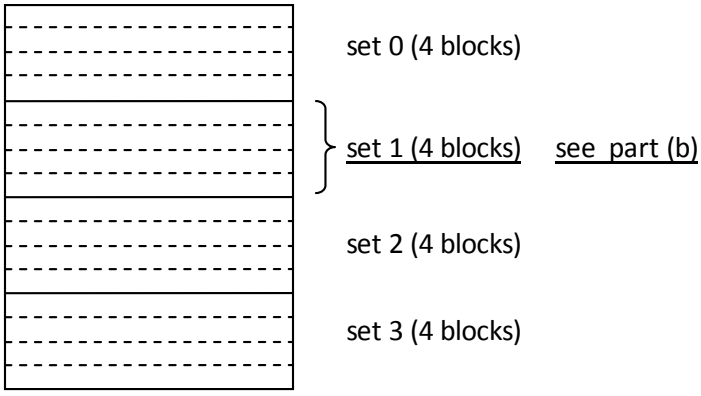
(3) Whether i > c or i <= c depends on multiprocessor cache design. If i <= c, then the answer in part (b) reduces to that in part (a). If i > c, then the second term involving f_{dir} reduces to $i*(1-f_i)h_d f_{dir}$.

5.12 (a) Given: 128 bytes in cache & 8 bytes per cache block. So $w = \log_2 8 = 3$.
→ Number of cache blocks = 128/8 = 16.
4-way set associative cache = 4 sets of 4 blocks each.
→ Set number $d = \log_2 4 = 2$.
Memory address size = 32 bits = $s + w$ with $s = 29$ and $w = 3$.
Tag field size = $s - d = 27$ bits. Thus the 32 address bits are divided as:



Cache organization:

Note: Each cache block has a 27 bit tag field associated with it, which is not shown.



(b) Given address = 000010**AF**$_{16}$
The last 8 bits of the address are AF$_{16}$ = 101**01**111 in binary.
Comparing with the diagram, set number d = **01** (binary) = 1.
The four cache blocks of Set 1 are shown in the diagram above.

(c) Given address = FFFF7Axy$_{16}$
The right-most eight bits are xy$_{16}$. Let this be $x_3x_2x_1\mathbf{x_0y_3}y_2y_1y_0$
In these, the set number should be 01 (binary), to match part (b).
Comparing with the memory address format, set number $\mathbf{x_0y_3}$ = 01.
The remaining 6 bits in xy$_{16}$ can (independently) be set to 0 or 1.
These bits are $x_3x_2x_1$ and $y_2y_1y_0$. The 64 possible values of these bits will map into the same cache set number.
Along with $\mathbf{x_0y_3}$ = 01, these 64 give the required values of hex digits x and y.
[Too many to list out explicitly!]

5.13 We must assume that the average memory access time *t* takes into account the delays caused by bus contention.

(a) Consider at first one processor.
MIPS rate based on local memory = *x* (given). Assume *t* is in μs.
→ Time to execute one instruction on local memory = $1/(x*10^6)$ sec = $1/x$ μs.
Extra memory access time (on average) spent on each instruction = *mt* (in μs).
Total time spent per instruction (with local+shared memory) = $1/x + mt$
→ MIPS rate will go down in that proportion.
Reduced MIPS rate
$= x * (1/x) / [1/x + mt]$
$= x/[1+xmt]$
[Note: Makes sense that the reduced rate does not depend on *n*.]
Since bus contention time is included in t, with p processors we get:

Reduced MIPS rate of p processors = $px/[1+xmt]$.

(b) Given $p = 32$, $m = 0.4$, $t = 0.01$ µs, nett MIPS = 5600.
→ $5600 = 32x/[1+0.004x]$
→ $1+0.004x = 32x/5600$ giving $x = \underline{583.3}$

Note that 32*583.3 = 18667.
The huge loss in performance (5600 compared to 18667) is evident, caused by the relatively frequent (m=0.4) access to shared memory.

(c) Now m is going up to 1.6! Presumably because of CISC processors?
Effective MIPS = 32*200/[1+200*1.6*0.01] = 6400/4.2 = $\underline{1524}$.
[Compare with figure of 5600 seen in (b).]

5.14 The basic expression of effective MIPS = $x/[1+xmt]$ seen above is also valid here, where t = additional memory access time (over and above local cache access).

Result of Problem 5.11:

$$t_{eff} = b*f_i(1-h_i) + c*f_ih_i + c*(1-f_i)h_d(1 - f_{dir}) + \max(c,i)*(1-f_i)h_df_{dir}$$
$$+2b*(1-f_i)(1-h_d)f_{dir} + b*(1-f_i)(1-h_d)(1-f_{dir})$$

From this, remove terms for local cache access and synchronization – which are the second, third and fourth terms, giving non-local access time t_{add} as:

$$t_{add} = b*f_i(1-h_i) + 2b*(1-f_i)(1-h_d)f_{dir} + b*(1-f_i)(1-h_d)(1-f_{dir})$$

Recall that 'b' here is the cache block access time. In the present problem, the average memory access time is given as t_m. Since there is no explicit reference to writing back dirty cache blocks, we replace both 'b' and '2b' by t_m, giving:

$$t_{add} = t_m*f_i(1-h_i) + t_m*(1-f_i)(1-h_d)f_{dir} + t_m*(1-f_i)(1-h_d)(1-f_{dir})$$

Last two terms combine to give:

$$t_{add} = t_m*f_i(1-h_i) + t_m*(1-f_i)(1-h_d)$$ [Actually quite straightforward!]
$$= t_m[f_i(1-h_i) + (1-f_i)(1-h_d)]$$

But we must also discount the effective MIPS by the overhead of synchronization (over and above local cache access), which is given by αt_s. Proceeding as before:

Nett MIPS rating of one processor = $x/[(1+xmt_{add})(1+x\alpha t_s)]$

For p processors, nett MIPS = $px/[(1+xmt_{add})(1+x\alpha t_s)]$ ← required expression

[As before, assuming effective memory access time includes bus contention.]

(b) $t_{add} = t_m[f_i(1-h_i) + (1-f_i)(1-h_d)]$
$= 0.05[0.5*0.05 + 0.5*0.2] = 0.05[0.025+0.1] = 0.05*0.125 = 0.00625$ μs

Thus nett MIPS of one processor

$= 500/[(1+500*0.4*0.00625)(1+500*0.02*1)]$

$= 500/[(1+1.25)(1+10)] = \underline{20.2}$

Note:

This is a very low MIPS figure, caused by the large values of α and t_s, which lead to the factor of 11 in the denominator. If $\alpha = 0.02$, and $t_s = 1$ μs, then actually once in every 50 instructions, we lose time worth 500 instructions (= 1 μs)! Not a good design!

Possibly the figure of t_s should be e.g. 0.01 μs (equivalent to 5 processor instructions, instead of 500). This would give effective MIPS = 500/2.25*1.1= 202.

Therefore for nett MIPS of 2000, number of processors p = 2000/20.2 = 99, say 100.

(c) Total cache cost = 16*96*1.25 = $ 1920/-.
Money left for shared memory = $ 25K - $ 1.92K = $ 23.08K
Memory in Kbytes which can be purchased = 23.08K/0.1 = 230.8K
→ In Mbytes, this memory equals 230.8 Mbytes.

CHAPTER 6 Selected Solutions

6.1 The relevant expressions are given in sub-section 6.1.3.

(a) Speedup factor $S_k = nk/(k+n-1) = 1.5x10^6x5/(4+1.5x10^6)$

= almost exactly k = 5.

[From the expression for S_k, we can see that as n goes to infinity, S_k goes to k. Here $1.5x10^6$ is a pretty large number. For smaller values of n, the value of S_k will be smaller. Students can easily verify.]

(b) Efficiency = $S_k/k = 1$ (that is, 100%). In the same way, with this large n, throughput = $1/\tau = f$. Students can also try smaller values of n.

6.3 The ratio will be maximized when its inverse is minimized.
[In this case it is easier to differentiate the inverse!]

On differentiating $(t/k+d)(c+kh)$ with respect to k, and equating to zero, we get:

$h(t/k+d) - (c+kh)t/k^2 = hd + ht/k - ct/k^2 - ht/k = 0$

This gives:

$ct/k^2 = hd$

Therefore $k_{opt} = sqrt(tc/hd)$ as required

6.5 (a) Forbidden latencies: 5 (from row 1), 2 (from row 2), 1 (from row 4)

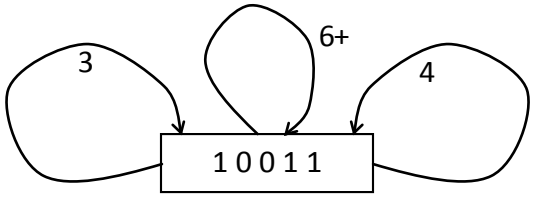
→ Initial collision vector: 10011 (3 and 4 are permissible latencies)

(b) Initial state: 10011

Shift right by 3 and then OR with initial state: 00010 OR 10011 = 10011
Shift right by 4 and then OR with initial state: 00001 OR 10011 = 10011

Note that, after latency of 3 or 4, we simply get back to initial state!

(b) State transition diagram:



(c) Thus MAL = 3. But MAL < 2 = the maximum number of tick-marks in any one row. (See the results on MAL bounds in sub-section 6.2.3)

(d) Throughput → One output in 3τ = 6 ns
→ 166.7 million operations per second (MOPS?).

(e) Lower bound on MAL is 2, which has NOT been obtained.

6.6 Introduce a non-compute delay stage between two consecutive tick-marks of S4, between old time slots 4 and 5, to enable permissible latency of 1. Therefore modified reservation table:

1	2	3	4	5	6	7
X						X
	X		X			
		X				
			X		X	

Forbidden latencies: 6, 2
Allowed latencies: 5, 4, 3, 1
Initial collision vector: 100010

See partial state transition diagram below.

A few key steps in deriving the diagram are shown here:
Shift 5 and OR with initial: 000001 OR 100010 → 100011
Shift 4 and OR with initial: 000010 OR 100010 → 100010 (initial state)
Shift 3 and OR with initial: 000100 OR 100010 → 100110
Shift 1 and OR with initial: 010001 OR 100010 → 110011

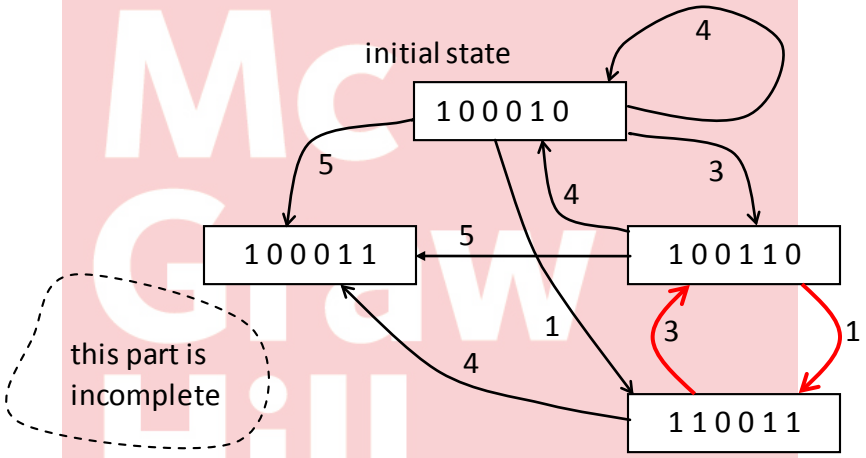
With four permissible latencies, there will be MANY reachable states.
However, to find MAL, we need a cycle having latency 1 as an edge.
[Any cycle NOT including latency 1 will have average latency > 2.]

This means we focus on states with '0' in the rightmost position.

From state: 100110
Shift 5 and OR with initial: 000001 OR 100010 → 100011
Shift 4 and OR with initial: 000010 OR 100010 → 100010 (initial)
Shift 1 and OR with initial: 010011 OR 100010 → 110011

From state: 110011
Shift 4 and OR with initial: 000011 OR 100010 → 100011
Shift 3 and OR with initial: 000110 OR 100010 → 100110

We see that, there is a cycle of length 2 between states 100110 and 110011, with edge latencies of 3 and 1, respectively. See partial diagram below (latency 7+ edges not shown):



To list all the simple cycles etc. the state transition diagram should be completed. This part can be left to the students, including the 7+ edges.

MAL = (3+1)/2 = 2 = optimum, according the bound given in sub-section 6.2.3 (because maximum number of crosses in a row = 2).

Two operations per 4τ, i.e. one operation per 4 ns on average.
→ 250 MOPS. This is a 50% improvement over Problem 6.5.

6.7 (a) To minimize the number of clock cycles, we need to keep the pipelined adder 'as busy as possible'. The following time-based tables show the process:

Here we are assuming N = 64 for the specified array size.

Notation: Aj:k in the table means the sum of A(j)+A(j+4)+….+A(k). That is, Aj:k is the sum of every fourth element of the array from A(j) to A(k) including both.

The two coloured cells indicate how A1:1 = A(1) is added to A(5) to give A1:5. That is, A(1) comes back to input Y of S1 while A(5) is fed on input X of S1. Four cycles later, in the same way, A1:5 + A(9) gives A1:9. This pattern is seen throughout.

t →	1	2	3	4	5	6	7	8	9	10	11	12
S4				A1:1	A2:2	A3:3	A4:4	A1:5	A2,6	A3:7	A4:8	A1:9
S3			A1:1	A2:2	A3:3	A4:4	A1:5	A2:6	A3:7	A4:8	A1:9	A2:10
S2		A1:1	A2:2	A3:3	A4:4	A1:5	A2:6	A3:7	A4:8	A1:9	A2:10	A3:11
S1	A1:1	A2:2	A3:3	A4:4	A1:5	A2:6	A3:7	A4:8	A1:9	A2:10	A3:11	A4:12

Time slots 13 to 56 are not shown; basically the same pattern seen above recurs. But the last eight time slots are shown below, so as to make the partial sums clear.

t →	57	58	59	60	61	62	63	64
S4	A2:54	A3:55	A4:56	A1:57	A2:58	A3:59	A4:60	A1:61=U
S3	A3:55	A4:56	A1:57	A2:58	A3:59	A4:60	A1:61	A2:62=V
S2	A4:56	A1:57	A2:58	A3:59	A4:60	A1:61	A2:62	A3:63=W
S1	A1:57	A2:58	A3:59	A4:60	A1:61	A2:62	A3:63	A4:64=X

At the end of 64 clock cycles, four partial sums are in stages S4…S1. We designate these by U, V, W, X. Now we must add these four to get the final array sum, as shown in the table below (time slot 64 is repeated for clarity). Note that, for adding the final four partial sums, register stage R is also used.

t	64	65	66	67	68	69	70	71	72	73	74	75
R		U		W			U:V	U:V				
S4	U	V	W	X		U:V		W:X				**U:X**
S3	V	W	X		U:V		W:X				U:X	^ final output
S2	W	X		U:V		W:X				U:X		
S1	X		U:V		W:X				U:X			

Final output = U:X = U+V+W+X = A1:61 + A2:62 + A3:63 + A4:64.
Total time slots needed = 75 = 64+11.

This result is in fact easy to generalize for array A of size N, provided that N is a multiple of 4 (the number of pipeline stages).

Therefore answer to part (a) is: number of cycles needed = N+11.

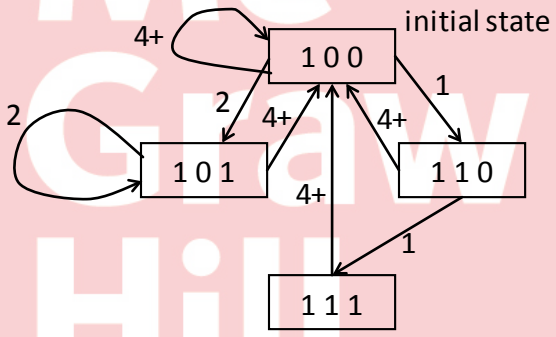
(b) Speedup = 64x4τ/75τ = 256/75 = 3.41
Efficiency = 3.41/4 = 0.853 → 85.3%

(c) As N tends to infinity, the extra 11 cycles don't really count! So then speedup goes to its maximum possible value 4, and efficiency goes to 100%.

(d) For half the maximum speedup, speedup equals 4/2 = 2.
Therefore 4N/(N+11) = 2. Thus 2N = N+11, i.e. N = 11.

6.8 (a) Forbidden latency: 3
(b) Initial collision vector = 100 → initial state
From initial state:
After p=1 shift and OR: 110
After p=2 shifts and OR: 101
From state 110:
After p=1 shift and OR: 111
From state 101:
After p=2 shifts and OR: 101

Diagram:



(c)
Simple cycles: (2,4+), (1,4+), (1,1,4+), (2), (4+)
Greedy cycles: (1,1,4+), (2)

(d) Optimum constant latency cycle: (2). MAL = 2.
(e) Clock period = 2 ns → latency = 2x2 = 4 ns → throughput = 250 MHz.

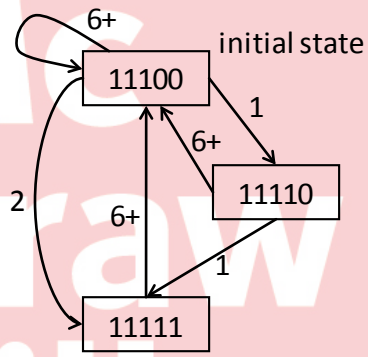
6.9 (a) Forbidden latencies: 3, 4 and 5
Initial collision vector: 11100 = initial state.

(b)
From initial state:
Shift p = 2 and OR: 11111
Shift p = 1 and OR: 11110

From state 11111: Only latency 5+ is possible.

From state 11110:
Shift p = 1 and OR: 11111

State transition diagram:



Note: By definition, all paths and cycles seen in the diagram correspond to latency sequences which do not cause any collision in the pipeline.

(c) Simple cycles: (1,6+), (1,1,6+), (2,6+), (6+)

(d) Greedy cycle: (1,1,6+)

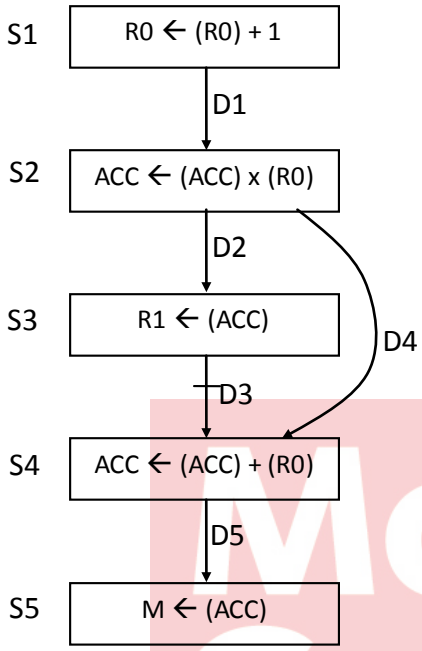
(e) MAL = (1+1+6)/3 = 2.67

(f) Minimum allowed constant cycle: (6+) i.e. latency 6 from initial state.

(g) Throughput = inverse of MAL = $1/(2.67\tau)$ for clock period τ.

(h) Minimum constant cycle is (6+), giving throughput = $1/(6\tau)$.

6.10 The following diagram shows data dependences in the given code:



Note: (1) Flow dependence from S1 to S4 is not shown, because the two from S1 to S2, and from S2 to S4, are present. (2) Self-dependence of S4 is not shown for clarity; it has no bearing on the discussion below.

[Dependences are numbered from D1 to D5. Meaning of arrow notation is given in Chapter 1.]

Now we schedule the instructions in the 3-stage processor pipeline:

time slot →	1	2	3	4	5	6	7
Fetch	S1	S2	S3	S4	S5		
Op fetch		S1	S2	S3	S4	S5	
Execute			S1	S2	S3	S4	S5

It is seen that the simple 'straight-through' scheduling of the five instructions does not violate any of the dependences, i.e. create any hazards.

Hazards are avoided because instruction execution occurs only in stage 3 of the pipeline. All the input operands of S1-S5 are in processor registers, and all operations take 1 clock cycle. Therefore if the five instructions go through S3 in order, no hazard is created.

[It has become an easy problem to solve, but that is only because the three-stage instruction pipeline specified has a simple structure!]

6.11 Reservation table for the given pipeline is as under:

S1	X				X	
S2		X		X		X
S3			X			
S4				X		

[It seems to be the case, from the way that the problem is specified, that output is NOT to be taken after the second execution of S2, but after the third. The second execution of S2 must have a role (?) and therefore it is presumably needed for the correct output after the third execution of S2.]

The rest of the problem is very similar to MAL problems seen earlier.

6.12 Reservation table is completed as under:

	1	2	3	4	5	6	7	8	9	10	11	12
S1	X								X			
S2		X								X		
S3			X	X							X	X
T1					X			X				
T2						X						
T3							X					
U1					X		X					
U2								X				
U3						X						
	f1 operating				f2, f3 operating				f1 operating			

[Reservation tables of f1, f2, f3 have been inserted as per the given block diagram.]

The rest of the problem is very similar to MAL problems seen earlier.
However drawing the diagram requires much more work – but possibly nothing new would be learned from working through the huge number of states!

6.13 (a) On the non-pipelined processor:

Time of execution = 1000x4τ where τ is the clock period at 250 MHz.

On the pipelined processor:
Extra 4 (= n-1) clock cycles spent initially to fill the pipeline.

Time of execution = 1004xτ' where τ' = 1.25τ because of reduced clock rate.

Therefore speedup = 4000τ/(1004x1.25τ) = 4000/1255 = 3.19

(b) Time of execution on non-pipelined processor = 4000τ = 4000/250 μs = 16 μs.
Therefore MIPS rate = 1000 instructions/16 μs → 62.5 MIPS.

Time of execution on pipelined processor = 1004τ' = 1004/200 μs = 5.02 μs.
Therefore MIPS rate = 1000 instructions/5.02 μs → 199.2 MIPS.

[Ratio = 199.2/62.5 = 3.19 as it should be.]