

SOLUTIONS TO PRACTICE PROBLEMS

COMPUTER ORGANIZATION AND
ARCHITECTURE
DESIGNING FOR PERFORMANCE
NINTH EDITION

WILLIAM STALLINGS

Copyright 2012: William Stallings

TABLE OF CONTENTS

Chapter 2	Computer Evolution and Performance.....	3
Chapter 3	Computer Function and Interconnection	7
Chapter 4	Cache Memory	9
Chapter 5	Internal Memory	14
Chapter 6	External Memory	16
Chapter 7	Input/Output	18
Chapter 8	Operating System Support.....	20
Chapter 9	Number Systems.....	22
Chapter 10	Computer Arithmetic.....	23
Chapter 11	Digital Logic.....	25
Chapter 12	Instruction Sets: Characteristics and Functions.....	26
Chapter 13	Instruction Sets: Addressing Modes and Formats ...	28
Chapter 14	Processor Structure and Function.....	29
Chapter 15	Reduced Instruction Set Computers	33
Chapter 16	Instruction-Level Parallelism and Superscalar Processors	36
Chapter 17	Parallel Processing	38
Chapter 18	Multicore Computers	40
Chapter 19	Control Unit Operation.....	41
Appendix B	Assembly Language and Related Topics	42

CHAPTER 2 COMPUTER EVOLUTION AND PERFORMANCE

2.1 This program is developed in [HAYE98]. The vectors A, B, and C are each stored in 1,000 contiguous locations in memory, beginning at locations 1001, 2001, and 3001, respectively. The program begins with the left half of location 3. A counting variable N is set to 999 and decremented after each step until it reaches -1. Thus, the vectors are processed from high location to low location.

Location	Instruction	Comments
0	999	Constant (count N)
1	1	Constant
2	1000	Constant
3L	LOAD M(2000)	Transfer A(I) to AC
3R	ADD M(3000)	Compute A(I) + B(I)
4L	STOR M(4000)	Transfer sum to C(I)
4R	LOAD M(0)	Load count N
5L	SUB M(1)	Decrement N by 1
5R	JUMP+ M(6, 20:39)	Test N and branch to 6R if nonnegative
6L	JUMP M(6, 0:19)	Halt
6R	STOR M(0)	Update N
7L	ADD M(1)	Increment AC by 1
7R	ADD M(2)	
8L	STOR M(3, 8:19)	Modify address in 3L
8R	ADD M(2)	
9L	STOR M(3, 28:39)	Modify address in 3R
9R	ADD M(2)	
10L	STOR M(4, 8:19)	Modify address in 4L
10R	JUMP M(3, 0:19)	Branch to 3L

2.2 Fraction of CPU which gets enhanced = $30/100=0.30$

Fraction of CPU in I/O activity = $60/100=0.60$

Speed up for enhancement given = 10

Therefore, overall speed up is = $1 / (0.50 + (0.30/10)) = 1.889$.

2.3 This question is about performance benchmarks for computer systems. It involves how rates of several programs of a benchmark suite are combined to give a single value.

- a.** To combine rates of execution one must use a weighted harmonic mean.

The WHM is defined as

$$WHM = \frac{1}{\sum_{i=1}^n \frac{W_i}{R_i}}$$

where W_i and R_i are the weights and rate for program i within the suite respectively.

One must not use an arithmetic mean or a simple harmonic mean, as these do not lead to a correct computation of the overall execution rate for the collection of programs. Each weight W_i is defined as the ratio of the amount of work performed in program i to the total amount of work performed in the whole suite of programs. Thus, if program i executes N_i floating-point operations then, assuming m programs in total;

$$W_i = \frac{N_i}{\sum_{j=1}^m N_j} \text{ or,}$$

$$WHM = \frac{N_1 + N_2 + N_3 + \dots + N_m}{t_1 + t_2 + t_3 + \dots + t_m} \text{ or,}$$

$$WHM = \frac{t_1 r_1 + t_2 r_2 + t_3 r_3 + \dots + t_m r_m}{t_1 + t_2 + t_3 + \dots + t_m}$$

We know that $T = \sum_{j=1}^m t_j$

Using the above two equations, we can say that

$$WHM = \frac{t_1}{T} r_1 + \frac{t_2}{T} r_2 + \dots + \frac{t_m}{T} r_m$$

- b.** For this example,

$WHM = (0.1 \times 1000 + 0.55 \times 110 + 0.2 \times 500 + 0.1 \times 200 + 0.05 \times 75)$, which is 284.25 mflops.

- c.** Consider P2 in isolation. If all other programs run at rate r_∞ then:

$$\lim_{r \rightarrow \infty} (WHM) = \frac{R_2}{W_2}$$

Thus the upper bound on the benchmark performance is simply R_2/W_2 . We therefore have $284:25=0:55 = 516:82$ mflops.

$$2.4 \quad \frac{(0.4 \times 15) + (1 - 0.4)}{0.4 + (1 - 0.4)} = 6.6$$

2.5 a. The design constraints are:

The constraints are that total system cost must be $\geq \$50k$ and $\leq \$100k$, and total system power consumption must be $< 50kW$.

The quantity or quantities to maximize is/are:

Since the goal of the product is to be a cost-performance leader, we'd

better try to maximize its cost-performance!

The quantity or quantities to minimize is/are:

System cost should be minimized, within the $\geq \$50k$ constraint.

b. (i) the constraints that our design must satisfy:

Constraints on total system cost:

$$C_{sys} = n_{chips} C_X \geq C_{sys,min} \quad C_{sys} = n_{chips} C_X \leq C_{sys,max}$$

Constraints on total system power:

$$P_{sys} = n_{chips} P_X \leq P_{sys,max}$$

(ii) the quantity that we should be maximizing in our design (figure of merit),

Maximize system cost-performance:

$$CP_{sys} = T_{sys}/C_{sys} = T_X n_{chips}/C_X n_{chips} = T_X/C_X.$$

(iii) the quantity that we should be minimizing in our design (figure of demerit)

Minimize system cost (within the minimum cost constraint):

$$C_{sys} = n_{chips} C_X = \lceil C_{sys,min}/C_X \rceil \times C_X$$

(This goal is secondary to minimizing the cost per unit of throughput, CX/TX , or maximizing cost-performance)

c. (i)

$$CP_{sys,A} = T_A/C_A = 80 \text{ w.u./s}/\$2.75k = 29.09 \text{ w.u./s}/\$1k$$

$$CP_{sys,B} = T_B/C_B = 50 \text{ w.u./s}/\$1.6k = 31.25 \text{ w.u./s}/\$1k$$

Chip B gives better cost-performance (31¼ work units per second per \$1,000 of cost, versus only 29.09 for chip A).

(ii) Since we want to minimize system cost $C_{sys,B} = n_{chips} C_B$, we must minimize n_{chips} , within the constraint that $n_{chips} C_B \geq \$50k$. Thus, the optimal (minimum) value of n_{chips} is given by

$$n_{chips,opt} = \lceil \$50k/C_B \rceil = \lceil \$50k/\$1.6k \rceil = \lceil 31.25 \rceil = 32.$$

(Since $32 \times 50W = 1.6kW < 50 kW$, the power constraint is met.)

(iii) If we had used chip A, then we would have had

$$n_{chips,opt} = \lceil \$50k/C_A \rceil = \lceil \$50k/\$2,750 \rceil = \lceil 18.18 \rceil = 19,$$

and

$$C_{sys,A} = 19 \times \$2,750 = \$52,250.$$

With chip B, we have

$$C_{sys,B} = 32 \times \$1,600 = \$51,200.$$

So yes, chip B also lets us achieve a design with lower total cost that still meets the \$50,000 minimum cost constraint.

Incidentally, although we didn't ask you to compute it, the optimal design with chip A would have a total throughput of 19×80 w.u./s = 1,520 work units per second, while the design with chip B will have a throughput of 32×50 w.u./s = 1,600 work units per second. So, the total throughput is also greater if we base our design on chip B. This is because the total cost of the chip-B-based design is about the same (near the \$50k target), but the cost-performance of chip B was greater.

- 2.6** Amdahl's law is based on a fixed workload, where the problem size is fixed regardless of the machine size. Gustafson's law is based on a scaled workload, where the problem size is increased with the machine size so that the solution time is the same for sequential and parallel executions.
- 2.7 a.** Say Program P1 consists of n x86 instructions, and hence $1.5 \times n$ MIPS instructions. Computer A operates at 2.5 GHz, i.e. it takes 0.4ns per clock. So the time it takes to execute P1 is $0.4\text{ns}/\text{clock} \times 2 \text{ clocks/instructions} \times 1.5 n \text{ instructions} = 1.2 n \text{ ns}$. Computer B operates at 3 GHz, i.e. 0.333ns per clock, so it executes P1 in $0.333 \times 3 \times n = n \text{ ns}$. So Computer B is 1.2 times faster.
- b.** Going by similar calculations, A executes P2 in $0.4 \times 1 \times 1.5 n = 0.6 \times n \text{ ns}$ while B executes it in $0.333 \times 2 \times n = 0.666 n \text{ ns}$ hence A is 1.11 times faster
- 2.8 a.** $\text{CPI} = .1 * 50 + .9 * 1 = 5.9$
- b.** $.1 * 50 * 100 / 5.9 = 84.75\%$
- c.** Divide would now take 25 cycles, so
 $\text{CPI} = .1 * 25 + .9 * 1 = 3.4$. Speed up = $5.9/3.4 = 1.735x$
- d.** Divide would now take 10 cycles, so
 $\text{CPI} = .1 * 10 + .9 * 1 = 1.9$. Speed up = $5.9/1.9 = 3.105x$
- e.** Divide would now take 5 cycles, so
 $\text{CPI} = .1 * 5 + .9 * 1 = 1.4$. Speed up = $5.9/1.4 = 4.214x$
- f.** Divide would now take 1 cycle, so
 $\text{CPI} = .1 * 1 + .9 * 1 = 1$. Speed up = $5.9/1 = 5.9x$
- g.** Divide would now take 0 cycles, so
 $\text{CPI} = .1 * 0 + .9 * 1 = .9$. Speed up = $5.9/.9 = 6.55x$

CHAPTER 3 COMPUTER FUNCTION AND INTERCONNECTION

3.1

Network Characteristics	Bus System	Multistage Network	Crossbar Switch
Minimum latency for unit data transfer	Constant	$O(\log_k n)$	Constant
Bandwidth per processor	$O(w/n)$ to $O(w)$	$O(w)$ to $O(nw)$	$O(w)$ to $O(nw)$
Wiring complexity	$O(w)$	$O(nw \log_k n)$	$O(n^2 w)$
Switching complexity	$O(n)$	$O(n \log_k n)$	$O(n^2)$
Connectivity and routing capability	Only one-to-one at a time	Some permutations and broadcast, if network unblocked	All permutations, one at a time
Remarks	Assume n processors on the bus; bus width is w bits	$n \times n$ MIN using $k \times k$ switches with line width of w bits	Assume $n \times n$ crossbar with line width of w bits

- 3.2 a.** Since the targeted memory module (MM) becomes available for another transaction 600 ns after the initiation of each store operation, and there are 8 MMs, it is possible to initiate a store operation every 100 ns. Thus, the maximum number of stores that can be initiated in one second would be:

$$10^9 / 10^2 = 10^7 \text{ words per second}$$

Strictly speaking, the last five stores that are initiated are not completed until sometime after the end of that second, so the maximum transfer rate is really $10^7 - 5$ words per second.

- b.** From the argument in part (a), it is clear that the maximum write rate will be essentially 10^7 words per second as long as a MM is free in time to avoid delaying the initiation of the next write. For clarity, let the module cycle time include the bus busy time as well as the internal processing time the MM needs. Thus in part (a), the module cycle time was 600 ns. Now, so long as the module cycle time is 800 ns or less, we can still achieve 10^7 words per second; after that, the maximum write rate will slowly drop off toward zero.

3.3 Total time for an operation is $10 + 15 + 25 = 50$ ns. Therefore, the bus can complete operations at a rate of 20×10^6 operations/s.

3.4 Interrupt 4 is handled first, because it arrives first. By the time the processor is done with interrupt 4, interrupts 7 and 1 are pending, so interrupt 1 gets handled. Repeating this process gives the following order: 4, 1, 0, 3, 2, 1, 4, 5, 6, 7.

3.5 a. 4k words of 8 bits each

b. LOW; LOW; LOW; inputs

c. The line from \overline{WR} to \overline{OE} plus the line from \overline{RD} to R/\overline{W}

CHAPTER 4 CACHE MEMORY

4.1 Let p be the probability that it is a cache hit. So $(1 - p)$ would be probability for cache miss. Now, $t_a = p \times t_h + (1 - p)t_m$. So

$$E = \frac{t_h}{t_a} = \frac{t_h}{p \times t_h + (1 - p)t_m} = \frac{1}{p + (1 - p)\frac{t_m}{t_h}}$$

- 4.2 a.** Using the rule $x \bmod 8$ for memory address x , we get $\{2, 10, 18, 26\}$ contending for location 2 in the cache.
- b.** With 4-way set associativity, there are just two sets in 8 cache blocks, which we will call 0 (containing blocks 0, 1, 2, and 3) and 1 (containing 4, 5, 6, and 7). The mapping of memory to these sets is $x \bmod 2$; i.e., even memory blocks go to set 0 and odd memory blocks go to set 1. Memory element 31 may therefore go to $\{4, 5, 6, 7\}$.
- c.** For the direct mapped cache, the cache blocks used for each memory block are shown in the third row of the table below

order of reference	1	2	3	4	5	6	7	8
block referenced	0	15	18	5	1	13	15	26
Cache block	0	7	2	5	1	5	7	2

For the 4-way set associative cache, there is some choice as to where the blocks end up, and we do not explicitly have any policy in place to prefer one location over another or any history with which to apply the policy. Therefore, we can imitate the direct-mapped location until it becomes impossible, as shown below on cycle 5:

order of reference	1	2	3	4	5	6	7	8
block referenced	0	15	18	5	1	13	15	26
Cache block	0	7	2	5	x			

It is required to map an odd-numbered memory block to one of $\{4, 5, 6, 7\}$, as shown in part (b).

- d.** In this example, nothing interesting happens beyond compulsory misses until reference 6, before which we have the following occupancy of the cache with memory blocks :

Cache block	0	1	2	3	4	5	6	7	V
-------------	---	---	---	---	---	---	---	---	---

Memory block	0	1	18			5		15	
--------------	---	---	----	--	--	---	--	----	--

On reference 6, memory block 13 takes the place of memory block 5 in cache block 5 (conflict miss), so memory block 5 goes to the victim cache:

Cache block	0	1	2	3	4	5	6	7	V
Memory block	0	1	18			13		15	5

The 7th reference, to memory block 15, hits in cache. On reference 8, memory block 26 is mapped to cache block 2, displacing memory block 18 to the victim cache and overwriting memory block 5:

Cache block	0	1	2	3	4	5	6	7	V
Memory block	0	1	26			13		15	18

Cycle 9 is a routine compulsory miss. On reference 10, memory block 18 is retrieved on the read miss from the victim cache, changing places with 26:

Cache block	0	1	2	3	4	5	6	7	V
Memory block	0	1	18			13		15	26

On reference 11, when memory block 5 is desired, it must come from memory, putting 13 into the victim cache:

Cache block	0	1	2	3	4	5	6	7	V
Memory block	0	1	18			5		15	13

Finally, on reference 12, memory block 13 is written into cache block 5. This does not reuse the old value of memory block 13 in the victim cache, though it does interchange 5 and 13 in cache block 5 and the victim cache upon completion.

To summarize, only reference 10 retrieves a victim cache block.

- 4.3 a.** Here a main memory access is a memory store operation. So we will consider both cases i.e. all stores are L1 miss and all stores are not L1 miss.

$$\begin{aligned}\text{All stores are not L1 miss} &= (\text{L1 miss rate}) \times (\text{L2 miss rate}) \\ &= (0.17) \times (0.12) = 2.04\%\end{aligned}$$

$$\begin{aligned}\text{All stores are L1 miss} &= (\% \text{ data references that read ops}) \times (\text{L2 miss rate}) \\ &+ (\% \text{ data ref that are writes}) \times (\text{L1 miss rate}) \times (\text{L2 miss rate}) \\ &= (0.50) \times (0.12) + (0.50) \times (0.17) \times (0.12) \\ &= .06 + .042 = .102 = 10.2\%\end{aligned}$$

- b.** Data = 8Kbytes/8 = 1024 bytes = 2^{10} = 10 bits

$$\text{Instruction} = 4\text{Kbytes}/8 = 512 \text{ bytes} = 2^9 = 9 \text{ bits}$$

$$\text{L2} = 2\text{M}/32 = 64\text{Kbytes} = 32\text{K sets} = 2^{15} = 15 \text{ bits}$$

- c.** Longest possible memory access will be when L1 miss + L2 miss + write back to main memory

$$\text{So, total cycles} = 1 + 10 + 2 \times 101 = 213 \text{ cycles.}$$

- d.** If you did not treat all stores as L1 miss then

$$\begin{aligned}(\text{Avg memory access time})_{\text{total}} &= (1/1.3)(\text{avg mem access time})_{\text{inst}} \\ &+ (.5/1.3)(\text{avg mem access time})_{\text{data}}\end{aligned}$$

Now,

$$\text{avg mem access time} = (\text{L1 hit time}) + (\text{L1 miss rate}) \times [(\text{L2 hit time}) + (\text{L2 miss rate}) \times (\text{mem transfer time})]$$

$$(\text{Avg memory access time})_{\text{inst}} = 1 + 0.02(10 + (0.10) \times 1.5 \times 101) = 1.503$$

$$(\text{Avg memory access time})_{\text{data}} = 1 + .17(10 + (0.10) \times 1.5 \times 101) = 5.276$$

$$\begin{aligned}(\text{Avg memory access time})_{\text{total}} &= (1/1.3)1.503 + (.5/1.3)5.276 \\ &= 1.156 + 2.02 = 3.18\end{aligned}$$

Note: We have to multiply the mean transfer time by 1.5 to account for the write backs in L2 cache. Now, let us treat all stores as L1 misses

$$\begin{aligned}(\text{Avg memory access time})_{\text{total}} &= (1/1.3)(\text{avg mem access time})_{\text{inst}} \\ &+ (.15/1.3)(\text{avg mem access time})_{\text{load}} + (.15/1.3)(\text{avg mem access time})_{\text{store}}\end{aligned}$$

$$\text{avg mem access time} = (\text{L1 hit time}) + (\text{L1 miss rate}) \times [(\text{L2 hit time}) + (\text{L2 miss rate}) \times (\text{mem transfer time})]$$

$$(\text{Avg memory access time})_{\text{inst}} = 1 + 0.02(10 + (0.10) \times 1.5 \times 101) = 1.503$$

$$(\text{Avg memory access time})_{\text{load}} = 1 + .17(10 + (0.10) \times 1.5 \times 101) = 5.276$$

$$(\text{Avg memory access time})_{\text{store}} = 1 + 1(10 + (0.10) \times 1.5 \times 101) = 26.15$$

$$\begin{aligned}(\text{Avg memory access time})_{\text{total}} &= (1/1.3)1.503 + (.15/1.3)5.276 + \\ &+ (.15/1.3)26.15 = 1.156 + .608 + 3.01 = 4.774\end{aligned}$$

Note: We have to multiply the mean transfer time by 1.5 to account for the write backs in L2 cache.

- 4.4** Write-allocate: When a write (store) miss occurs, the missed cache line is brought into the first level cache before actual writing takes place.
No Write-allocate: When a write miss occurs, the memory update bypasses the cache and updates the next level memory hierarchy where there is a hit.
- 4.5** Imagine a 4-word cache with an access pattern of 0, 1, 2, 3, 4, 0, 1, 2, 3, 4. The directed mapped cache will have a 30% hit rate while the LRU fully associative cache will have a 0% hit rate.
- 4.6** When a word is loaded from main memory, adjacent words are loaded into the cache line. Spatial locality says that these adjacent bytes are likely to be used. A common example is iterating through elements in an array.
- 4.7** The cache index can be derived from the virtual address without translation. This allows the cache line to be looked up in parallel with the TLB access that will provide the physical cache tag. This helps keep the cache hit time low in the common case of a TLB hit.
For this to work, the cache index cannot be affected by the virtual to physical address translation. This would happen if any of the bits from the cache index came from the virtual page number of the virtual address instead of the page offset portion of the address.
- 4.8** A case where LRU would be expected to outperform random replacement is a randomly-accessed tree structure that is too large to fit in the cache. The LRU policy will do a better job of keeping the root of the tree in the cache and those nodes of the tree are accessed more often.
- 4.9** One pathological case for LRU would be an application that makes multiple sequential passes through a data set that is slightly too large to fit in the cache. The LRU policy will result in a 0% cache hit rate, while under random replacement the hit rate will be high.
- 4.10** **a.** Can't be determined. $*(ptr + 0x11\ A538)$ is in the same set but has a different tag, it may or may not be in another way in that set
b. HIT. $*(ptr + 0x13\ A588)$ is in the same line as x (it is in the same set and has the same tag), and since x is present in the cache, it must be also
c. Can't be determined. It has the same tag, but is in a different set, so it may or may not be in the cache.

4.11 $EAT = .9 (10) + .1 [.8 (10 + 100) + .2 (10 + 100 + 10000)] = 220ns$
OR
 $EAT = 10 + .1 (100) + .02 (10000) = 220ns$

CHAPTER 5 INTERNAL MEMORY

5.1 First, we lay out the bits in a table:

Name	Address	Contents	C ₈	C ₄	C ₂	C ₁
M ₈	1100	0	0	0		
M ₇	1011	1	1		1	1
M ₆	1010	0	0		0	
M ₅	1001	0	0			0
C ₈	1000	1	1			
M ₄	0111	1		1	1	1
M ₃	0110	0		0	0	
M ₂	0101	1		1		1
C ₄	0100	1		1		
M ₁	0011	1			1	1
C ₂	0010	0			0	
C ₁	0001	1				1
P	0000	0				
Parities		1	0	1	1	1

As the global parity is incorrect, there is a one-bit error. The address of the error is 0111, so the corrected data is 0100001111010

5.2 The throughput of each bank is $(1 \text{ operation every } 100 \text{ ns})/8 = 12.5 \text{ ns}$, or $80 \times 10^6 \text{ operations/s}$. Because there are 4 banks, the peak throughput for the memory system is $4 \times 80 \times 10^6 = 320 \times 10^6 \text{ operations/s}$. This gives a peak data rate of $4 \text{ bytes} \times 320 \times 10^6 = 1.28 \times 10^9 \text{ bytes/s}$.

- 5.3 a.** Each bank can handle one operation/cycle, so the peak throughput is 2 operations/cycle.
- b.** At the start of each cycle, both banks are ready to accept requests. Therefore, the processor will always be able to execute at least one memory request per cycle. On average, the second memory request will target the same memory bank as the first request half the time, and will have to wait for the next cycle. The other 50% of the time, the two requests target different banks and can be executed simultaneously. Therefore the processor is able to execute on average 1.5 operations/cycle.
- c.** Each memory bank can execute 1 operation/cycle every 10 ns ($100 \times 10^6 \text{ operations/s}$). Therefore the peak data rate of each bank is $800 \times$

10^6 bytes/s. With two banks, the peak data rate is 1.6×10^9 bytes/s. The average data rate is $1.5 \times 800 \times 10^6$ bytes/s = 1.2×10^9 bytes/s.

- 5.4 a.** SRAMs generally have lower latencies than DRAMs, so SRAMs would be a better choice.
- b.** DRAMs have lower cost, so DRAMs would be the better choice.

CHAPTER 6 EXTERNAL MEMORY

6.1 The system requires at least one buffer the size of a sector, usually more than one. For systems with small memory, a smaller sector size reduces the memory needed for I/O buffers. The disk system requires a gap between sectors to allow for rotational and timing tolerances. A smaller sector size means more sectors and more gaps

6.2 A variable sector size complicates the disk hardware and disk controller design. In early systems it had the advantage of allowing for small block sizes with small memory systems, and larger block sizes for larger systems. Also, it allows the programmer to choose the optimal size for the data.

6.3 a. $T = \text{transfer time} = b/rN$, where b = number of bytes to transfer, r = rotation speed, and N = number of bytes on a track $\rightarrow b = 1\text{MByte}$
 $=$
 $1,048,576 \text{ Bytes}$, $r = 15,000\text{rpm}$, $N = 512 \times 400 = 204800 \text{ bytes per track} \rightarrow$
 $1,048,576 / (15,000 / 60,000 \times 204800) = 20.48\text{ms}$

Here are the units:

$$T[\text{ms}] = b[\text{bytes}] / (r[\text{rotations/ms}] \times N[\text{bytes/rotation}])$$

$$15,000[\text{rotations/min}] / 60,000[\text{ms/min}] = 0.25[\text{rotations/ms}]$$

Comment: This calculation is a simplified estimation but sufficient for basic disk transfer models: A more sophisticated estimate might notice that the file spans several tracks, thus a worst-case estimate is to include additional seek (but assume the track starts are staggered so there is no significant rotational delay).

- b.** $T_a = \text{average access time of the whole file} = T_s + 1/2r + b/rN = 4 + 2 + 20.508 \text{ ms} = 26.508\text{ms}$
- c.** Rotational delay = (average 180 degree wait) = $1/2r = 2\text{ms}$
- d.** Total time to read 1 sector (512 Bytes) = seek time + rotational delay
 $+ \text{transfer time} = 4\text{ms} + 2\text{ms} + 512 / (15000 / 60000 \times 204,500) = 4 + 2 + 0.01\text{ms} = 6.01\text{ms}.$
- e.** If the disk uses sequential organization, then the total time to read 1 track (400 sectors per track) = seek_time + rotational_delay + additional_time_to_go_around = $4\text{ms} + 2\text{ms} + 4\text{ms} = 10\text{ms} = (T_{\text{seek}} + 3 / (2r))$

6.4 $512 \text{ bytes} \times 1024 \text{ sectors} = 0.5 \text{ MB/track}$. Multiplying by 2048 tracks/platter gives 1 GB/platter, or 5 GG capacity in the drive.

CHAPTER 7 INPUT/OUTPUT

7.1

Memory Mapped	
Advantages	Disadvantages
Simpler hardware	Reduces memory address space
Simpler instruction set	Complicates memory protection
All address modes available	Complicates memory timing
I/O Mapped	
Advantages	Disadvantages
Additional address space	Makes small systems bigger
Easier to virtualize machine	Complicates virtual memory
	Complicates future expansion of address space

7.2 The first factor could be the limiting speed of the I/O device; Second factor could be the speed of bus, the third factor could be no internal buffering on the disk controller or too small internal buffering space. Fourth factor could be erroneous disk or transfer of block.

7.3 Prefetching is a user-based activity, while spooling is a system-based activity. Comparatively, spooling is a much more effective way of overlapping I/O and processor operations. Another way of looking at it is to say that prefetching is based upon what the processor might do in the future, while spooling is based upon what the processor has done in the past.

- 7.4 a.** The device makes 150 requests, each of which require one interrupt. Each interrupt takes 12,000 cycles (1000 to start the handler, 10,000 for the handler, 1000 to switch back to the original program), for a total of 1,800,000 cycles spent handling this device each second.
- b.** The processor polls every 0.5 ms, or 2000 times/s. Each polling attempt takes 500 cycles, so it spends 1,000,000 cycles/s polling. In 150 of the polling attempts, a request is waiting from the I/O device, each of which takes 10,000 cycles to complete for another 1,500,000 cycles. Therefore, the total time spent on I/O each second is 2,500,000 cycles with polling.

- c. In the polling case, the 150 polling attempts that find a request waiting consume 1,500,000 cycles. Therefore, for polling to match the interrupt case, an additional 300,000 cycles must be consumed, or 600 polls, for a rate of 600 polls/s.

7.5 Without DMA, the processor must copy the data into the memory as the I/O device sends it over the bus. Because the device sends 10 MB/s over the I/O bus, which has a capacity of 100 MB/s, 10% of each second is spent transferring data over the bus. Assuming the processor is busy handling data during the time that each page is being transferred over the bus (which is a reasonable assumption because the time between transfers is too short to be worth doing a context switch), then 10% of the processors time is spent copying data into memory.

With DMA, the processor is free to work on other tasks, except when initiating each DMA and responding to the DMA interrupt at the end of each transfer. This takes 2500 cycles/transfer, or a total of 6,250,000 cycles spent handling DMA each second. Because the processor operates at 200 MHz, this means that 3.125% of each second, or 3.125% of the processor's time, is spent handling DMA, less than 1/3 of the overhead without DMA.

7.6 memory-mapped.

CHAPTER 8 OPERATING SYSTEM SUPPORT

- 8.1** A specially tailored hardware designed for accelerating address translation from virtual address space to physical address space
- 8.2**
- a.** Since it has a 4K page size, the page offset field is the least significant 12 bits of the address (as 212 is 4K). So it contains 0x234.
 - b.** As the TLB has 16 sets, four bits are used to select one of them (both students taking the exam wanted to know how many entries the TLB has, and how associative it was. That information wasn't necessary to solve the question: the width of the TLB set index field is given by the number of sets. The number of entries and the associativity is used to calculate the number of sets. The next four bits in the address are 0x1
 - c.** The remaining bits in the address have to match the TLB tag. So that's 0xabcd
 - d.** The line size is sixteen bytes, so the offset field is the least significant four bits. So it contains 0x4.
 - e.** The cache is 32K and has a 16 byte line, so it has 2K lines. Since it's 8-way set-associative, it has 256 sets. This means there are eight bits in the set index field; it contains 0x23
 - f.** 0x1247
 - g.** Yes. None of the bits used in the cache lookup are changed by the virtual/physical translation.
- 8.3** Two examples are:
- 1. isolation: protect processes from each other's memory.
 - 2. relocation: allow any program to run anywhere in physical memory.
- 8.4**
- a.** It is addressed by virtual address.
 - b.** offset = 4 bits
index = 12 bits
tag = 16 bits
 - c.** 12 bit offset => 20-bit VPN
 - d.** 4 bits
- 8.5**
- a.** The time required to execute a batch is $M + (N \times T)$, and the cost of using the processor for this amount of time and letting N users wait meanwhile is $(M + (N \times T)) \times (S + (N \times W))$. The total cost of service time and waiting time per customer is

$$C = (M + (N \times T)) \times (S + (N \times W))/N$$

The result follows by setting $dC/dN = 0$
b. \$0.60/hour.

- 8.6** The countermeasure taken was to cancel any job request that had been waiting for more than one hour without being honored.
- 8.7** The problem was solved by postponing the execution of a job until all its tapes were mounted.
- 8.8** An effective solution is to keep a single copy of the most frequently used procedures for file manipulation, program input and editing permanently (or semi-permanently) in the internal store and thus enable user programs to call them directly. Otherwise, the system will spend a considerable amount of time loading multiple copies of utility programs for different users.

CHAPTER 9 NUMBER SYSTEMS

9.1

- | | |
|----------|---------|
| a. 1096 | e. 3666 |
| b. 446 | f. 63 |
| c. 10509 | g. 63 |
| d. 500 | h. 63 |

9.2 :

- | | |
|----------|------------|
| a. 3123 | d. 1022200 |
| b. 200 | e. 1000011 |
| c. 21520 | |

- 9.3**
- a. Going from most significant digits to least significant, we see that 23 is greater than 1, 3, and 9, but not 27. The largest value that can be represented in three digits, 111, is decimal 13, so we will need at least four digits. 1000 represents decimal 27, and 23 is four less than 27, so the answer is 10 $\perp\perp$
 - b. 99 is greater than 81 by 18, and 18 is twice 9. We can't put 2 in the nines place, but instead use 27 – 9 so the answer is 11 \perp 00 . That is: 81 + 27 – 9
 - c. We know 23 from (a). Negate by changing the sign of the digits for \perp 011
- 9.4** As with any other base, add one to the ones place. If that position overflows, replace it with the lowest valued digit and carry one to the next place.

0, 1, 1 \perp , 10, 11, 1 $\perp\perp$, 1 \perp 0, 1 \perp 1, 10 \perp , 100, 101, 11 \perp , 110, 111, 1 $\perp\perp\perp$, 1 $\perp\perp$ 0, 1 $\perp\perp$ 1, 1 \perp 0 \perp , 1 \perp 00, 1 \perp 01, 1 \perp 1 \perp

CHAPTER 10 COMPUTER ARITHMETIC

$$\begin{aligned}
 10.1 \quad -30.375 &= (-11110.011)_{\text{binary}} \\
 &= (-1.1110011)_{\text{binary}} \times 2^4 \\
 &= (-1)^1 \times (1 + 0.1110011) \times 2^{(131-127)} \\
 &= (-1)^{\text{sign}} \times (1 + \text{fraction}) \times 2^{(\text{exponent}-127)}
 \end{aligned}$$

$$\text{sign} = 1$$

$$\text{exponent} = 131 = (1000\ 0011)_{\text{binary}}$$

$$\text{fraction} = (1110\ 0110\ 0000\ 0000\ 0000\ 000)_{\text{binary}}$$

$$\begin{aligned}
 (-30.375)_{10} &= (\text{sign exponent fraction})_{\text{binary}} \\
 &= (1\ 1000\ 0011\ 1110\ 0110\ 0000\ 0000\ 0000\ 000)_{\text{binary}} \\
 &= (1100\ 0001\ 1111\ 0011\ 0000\ 0000\ 0000\ 0000)_{\text{binary}} \\
 &= (\text{C1F3}\ 0000)_{16}
 \end{aligned}$$

$$\begin{aligned}
 10.2 \quad A &= 0\ 01111111\ 11000\dots0 = 1.75 \times 2^0 \\
 B &= 1\ 10000001\ 00000\dots0 = -4 \times 2^0 \\
 C &= -7 = -1.11_2 \times 2^2 \\
 \text{Single-precision product} &= 0\text{x}\text{C0E00000}
 \end{aligned}$$

10.3 a.

$$\begin{array}{r}
 999 \\
 -123 \\
 \hline
 876
 \end{array}$$

b.

$$\begin{array}{r}
 999 \\
 -000 \\
 \hline
 999
 \end{array}$$

c.

$$\begin{array}{r}
 999 \\
 -777 \\
 \hline
 222
 \end{array}$$

10.4 Notice that for a normalized floating point positive value, with a binary exponent to the left (more significant) and the significand to the right can be compared using integer compare instructions. This is especially

useful for sorting, where data can be sorted without distinguishing fixed point from floating point. Representing negative numbers as the two's complement of the whole word allows using integer compare for all floating point values.

It might complicate floating point hardware, though without trying to actually design such hardware it is hard to know for sure.

10.5 a. -5 b. +15 c. -113 d. +119

10.6 a. 11111011 b. 11011000 c. 00000011 d. 00101011

10.7 a. $(0.2233 \times 10^2) + (0.6688 \times 10^1) = 0.2901 \times 10^2$
b. $(5.666 \times 10^0) + (44.55 \times 10^0) = 0.5021 \times 10^2$
c. $(111.77 \times 10^0) + (55.666 \times 10^0) = 0.1673 \times 10^3$

10.8 a. $(0.9922 \times 10^{-3}) - (0.4477 \times 10^{-3}) = 0.5445 \times 10^{-3}$
b. $(33.666 \times 10^0) - (2.7777 \times 10^0) = 0.3088 \times 10^2$
c. $(0.8888 \times 10^2) - (0.2222 \times 10^3) = -0.1333 \times 10^3$

10.9 a. $(0.5432 \times 10^3) \times (0.3333 \times 10^{-5}) = 0.1810 \times 10^{-2}$
b. $(222.88 \times 10^0) \times (1.1177 \times 10^0) = 0.2488 \times 10^3$

10.10 a. $(0.2233 \times 10^{-2}) / (0.6611 \times 10^3) = 0.3377 \times 10^{-5}$
b. $(111.99 \times 10^0) / (44.888 \times 10^0) = 0.2493 \times 10^1$

CHAPTER 11 DIGITAL LOGIC

11.1

Op signals			1 bit ALU function
binv	op1	Op0	
0	0	0	$r = a \text{ and } b$
0	0	1	$r = a \text{ or } b$
0	1	0	$\text{co}, r = \text{ci} + a + b$
0	1	1	$r = a \text{ xor } b$
1	0	0	$r = a \text{ and } b'$
1	0	1	$r = a \text{ or } b'$
1	1	0	$\text{co}, r = \text{ci} + a - b$
1	1	1	$r = a \text{ xor } b$

CHAPTER 12 INSTRUCTION SETS: CHARACTERISTICS AND FUNCTIONS

12.1

Little Endian	Low memory address	Big Endian
FE	← 0x10010000 →	21
FF	← 0x10010001 →	2A
2A	← 0x10010002 →	FF
21	← 0x10010003 →	FE
43	← 0x10010004 →	00
80		0C
0C		80
00		43
1B		01
00		94
94		00
01		1B
05		15
00		E0
E0		00
15	← 0x1001000F →	05
	High memory address	

12.2 From Table F.1 in Appendix F, 'z' is hex X'7A' or decimal 122; 'a' is X'61' or decimal 97; $122 - 97 = 25$

12.3 Let us consider the following: Integer divide $5/2$ gives 1, $-5/2$ gives -2 on most machines.

Bit shift right, 5 is binary 101, shift right gives 10, or decimal 2.

In eight bit twos complement, -5 is 11111011. Arithmetic shift right (shifting copies of the sign bit in) gives 11111101 or decimal -3.

Logical shift right (shifting in zeros) gives 01111101 or decimal 125.

- 12.4 a.** Decimal 5 in eight bit binary is 00000101, the ones complement of that being 11111010 for -5. Arithmetic right shift gives 11111101 or decimal -2.
- b.** Decimal 6 is 00000110, so -6 is 11111001. Arithmetic right shift gives 11111100 or -3

12.5 The nines complement of 123 is formed by subtracting each digit from 9, resulting in $999 - 123 = 876$.

876 is the three digit nines complement representation for -123.

$$\begin{array}{r} 456 \\ +876 \\ \hline 1332 \end{array}$$

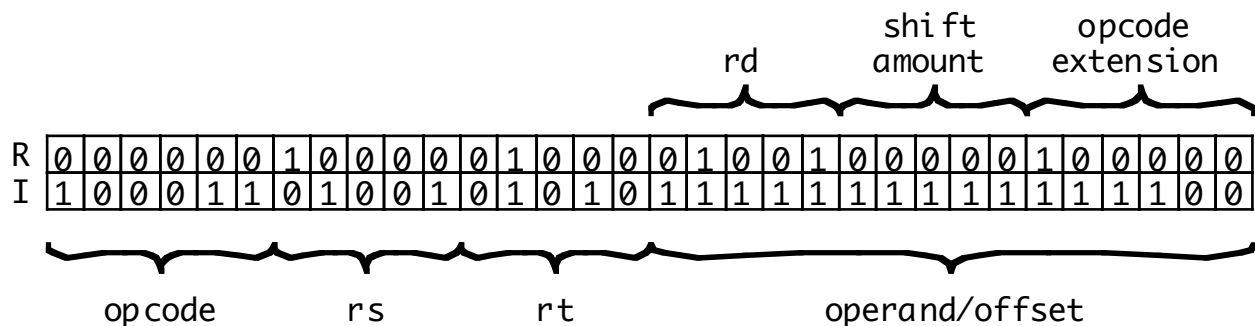
But we want a three digit result. In nines complement arithmetic it is necessary to do end-around carry, and add the carry back resulting in 333, the expected answer for $456 - 123$.

Many decimal subtraction problems can be done easily in your head using nines complement addition. You might even try it while balancing your checkbook.

CHAPTER 13 INSTRUCTION SETS: ADDRESSING MODES AND FORMATS

- 11.3 a.** \$20 = 0xFFFF FFFC
 \$20 = 1111 1111 1111 1111 1111 1111 1111 1100
- b.** \$20 = 0x0014 0000
 \$20 = 0000 0000 0001 0100 0000 0000 0000 0000
- c.** \$20 = 0x8FFF FFF2
 \$20 = 1000 1111 1111 1111 1111 1111 1111 0010
- d.** \$20 = 0x0000 0000
 4294967190 >= 17 => \$20=0
- e.** \$20 = 0x0000 0000
 4294967190 >= 20 => \$20=0
- f.** \$20 = 0x0000 0001
 -106 < -20 => \$20=1
- g.** \$20 = 0x0000 0001
 -106 < 17 => \$20= 1

11.3



0x02084820, 0x8d2afffc

CHAPTER 14 PROCESSOR STRUCTURE AND FUNCTION

14.1 The average instruction execution time on an unpipelined processor is

$$T_u = \text{clock cycle} \times \text{Avg. CPI}$$

$$T_u = 1 \text{ ns} \times ((0.5 \times 4) + (0.35 \times 5) + (0.15 \times 4))$$

$$T_u = 4.35 \text{ ns}$$

The avg. instruction execution time on pipelined processor is

$$T_p = 1 \text{ ns} + 0.2 \text{ ns} = 1.2 \text{ ns}$$

$$\text{So speed up} = 4.35/1.2 = 3.625$$

14.2 Relaxed consistency model allows reads and writes to be executed out of order. The three sets of ordering are:

- W→R ordering
- W→W ordering
- R→W and R→R ordering

14.3 Control hazard: Instruction fetch depends on some in-flight instruction being executed. For example, the target address of a branch or jump is not immediately available after the branch / jump exits from fetch stage.

Solutions:

1. Design a hazard unit to detect the hazard and stall the pipeline.
2. branch prediction.
3. using delay slot in the instruction scheduling

14.4 a. Here is a possible sequence of instructions

```
call A
branch mispredict
call B (bumps the RAS pointer)
rollback to the branch
```

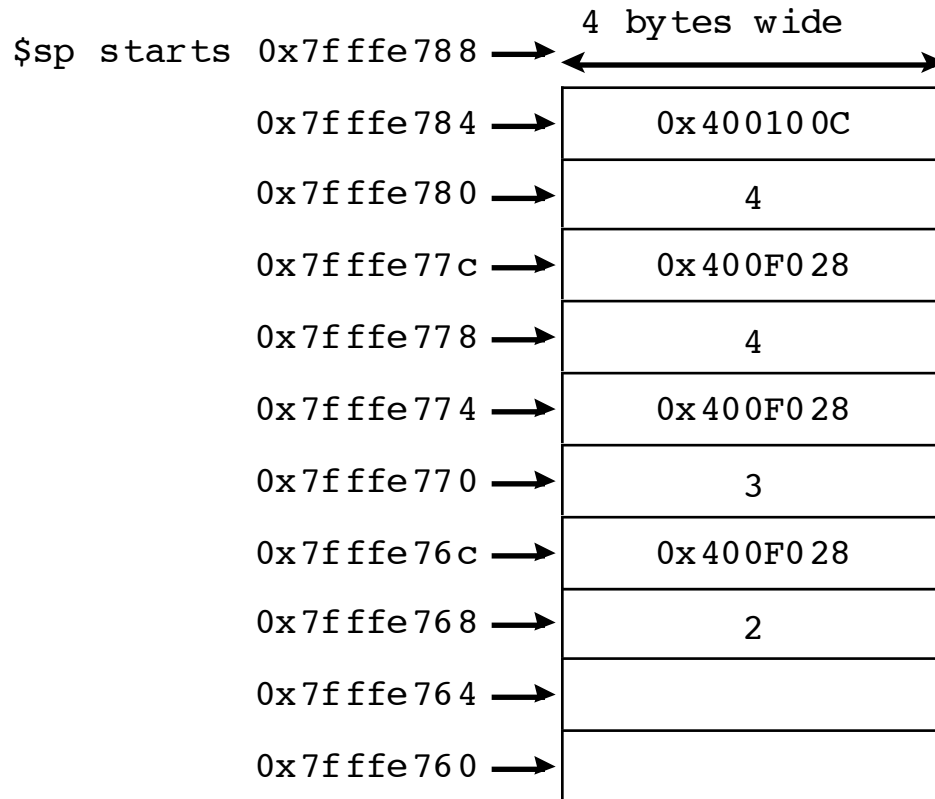
Now no matter what we do every return will get its target from the RAS entry one above where it should be looking

b. Here is the sequence of instructions.

```
A: call (headpointer <= 1, write value A+4 in RAS[1])
Fetch Branch that will mispredict (record headpointer = 1)
Return (headpointer <= 0)
```

B: call (headpointer ≤ 1 , and we overwrite $RAS[1] \leq B+4$)
 Rollback to the branch that mispredicted. This will restore the head pointer ≤ 1 , but $RAS[1] = B+4$, not $A+4$. Thus the next return will try to jump to $B+4$ and end up mispredicted.

14.5 a.



b. \$a0 = 10 decimal

14.6 a. This branch is executed a total of ten times, with values of i from 1 to 10. The first nine times it isn't taken, the tenth time it is. The first and second level predictors both predict correctly for the first nine iterations, and mispredict on the tenth (which costs two cycles). So, the mean branch cost is $2/10$ or 0.2

b. This branch is executed ten times for every iteration of the outer loop. Here, we have to consider the first outer loop iteration separately from the remaining nine. On the first iteration of the outer loop, both predictors are correct for the first nine inner loop iterations, and both mispredict on the tenth. So on the first outer loop iteration, we have a cost of 2 in the ten iterations. On the remaining iterations of the outer loop, the first-level predictor will mispredict on the first inner loop iteration, both predictors will be correct for the next eight iterations, and both will mispredict on the

final iteration, for a cost of 3 in ten branches. The total cost will be $2 + 9 \times 3$ in 100 iterations, or 0.29

- c. There are a total of 110 branch instructions, and a total cost of 31. So the mean cost is $31/110$, or .28

14.7 a.

	1	2	3	4	5	6
S1	X				X	
S2		X				X
S3			X			
S4				X		

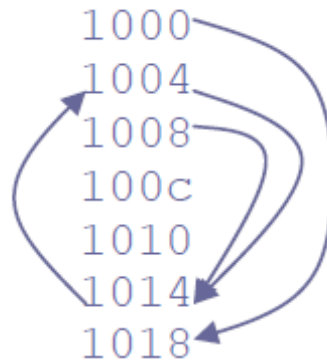
- b. {4}

- c. Simple cycles: (1,5), (1,1,5), (1,1,1,5), (1,2,5), (1,2,3,5), (1,2,3,2,5), (1,2,3,2,1,5), (2,5), (2,1,5), (2,1,2,5), (2,1,2,3,5), (2,3,5), (3,5), (3,2,5), (3,2,1,5), (3,2,1,2,5), (5), (3,2,1,2), and (3).

The greedy cycles are (1,1,1,5) and (1,2,3,2)

- d. $MAL = (1+1+1+5)/4 = 2$

14.8 a.



1004 taken to 1014 taken to 1004;
 1000 not taken to 1004 not taken to 1008;
 Both paths hash to table entry 2.

- b. 1000 not taken to 1004, 1004 taken to 1014
 1004 not taken to 1008, 1008 taken to 1014

14.9 $\text{CPI} = (\text{base CPI}) + (\text{CPI due to branch hazards}) + (\text{CPI due to FP hazards}) + (\text{CPI due to instruction accesses}) + (\text{CPI due to data accesses})$

Component	Explanation	Value
Base	Given in problem	1
Branch	branch % * misprediction % * misprediction penalty	$0.1 * 0.2 * 1 = 0.02$
FP	FP % * avg penalty	$0.2 * 0.9 = 0.18$
Instructions	AMAT for instructions minus 1 to take out the 1 cycle of L1 cache hit that is included in the base CPI	$1.476 - 1 = 0.476$
Data	(load % + store %) * AMAT for data adjusted as above	$(0.25 + 0.15) * (3.928 - 1) = 1.171$

This gives us a total CPI of 2.847

CHAPTER 15 REDUCED INSTRUCTION SET COMPUTERS

15.1 First draw the pipeline for this

-	1	2	3	4	5	6	7
A	X				X		
B		X		X			
C			X				X
D						X	

Intervals that cause collisions are

- Row A – 4,
- Row B – 2 ,
- Row C – 4, and
- Row D – none.

Therefore, the initial collision vector is - 001010

No. of 1's in the initial collision vector = 2.

Therefore, minimum average latency $\leq 2+1 = 3$

That is, maximum instruction throughput $\geq 1/3$ instructions per cycle.

Maximum number of checks in a row of the reservation table = 2

Therefore, minimum average latency ≥ 2

That is, maximum instruction throughput $\leq 1/2$ instructions per cycle

15.2 Buses

Instruction[15-0] after sign extension = 0x FFFF FFF4 (32-bit)

Read data 1 = 0x 0000 001D (32-bit)

Read data 2 = 0x 0000 0012 (32-bit)

Write Address in Data Memory = 0x 0000 0011 (32-bit)

Control signals

ALUSrc = 1

MemtoReg = X

PCSrc = 0

MemWrite = 1

RegDst = X

RegWrite = 0

15.3 a. The second instr. is dependent upon the first (because of \$2)

The third instr. is dependent upon the first (because of \$2)

The fourth instr. is dependent upon the first (because of \$2)

The fourth instr. is dependent upon the second (because of \$4)

There, All these dependencies can be solved by forwarding

- b.** Registers \$11 and \$12 are read during the fifth clock cycle.
Register \$1 is written at the end of fifth clock cycle.

15.4 a. Expected instructions to next mispredict is $e = 10/(1 - p)$. Number of instruction slots (two slots/cycle) to execute all those and then get the next good path instruction to the issue unit is $t = e + 2.20 = 10/(1 - p) + 40 = (50 - 40p)/(1 - p)$. So the utilization (rate at which we execute useful instructions) is $e/t = 10/(50 - 40p)$

- b.** If we increase the cache size the branch mispredict penalty increases from 20 cycles to 25 cycles than, by part (a), utilization would be $10/(60 - 50p)$. If we keep the i-cache small, it will have a non-zero miss-rate, m . We still fetch e instructions between mispredicts, but now of those e , em are 10 cycle (=20 slot) L1 misses. So the number of instruction slots to execute our instructions is

$$t = e(1 + 20m) + 2.20 = [(10)/(1 - p)][1 + 20m] + 4 \\ = (50 + 200m - 40p)/(1 - p)$$

The utilization will be $10/(50 + 200m - 40p)$. The larger cache will provide better utilization when

$$(10)/(60 - 50p) > (10)/(50 + 200m - 40p), \text{ or} \\ 50 + 200m - 40p > 60 - 50p, \text{ or} \\ p > 1 - 20m$$

15.5 Performance can be increased by:

- Fewer loop conditional evaluations.
- Fewer branches/jumps.
- Opportunities to reorder instructions across iterations.
- Opportunities to merge loads and stores across iterations.

Performance can be decreased by:

- Increased pressure on the I-cache.
- Large branch/jump distances may require slower instructions

15.6 The advantages stem from a simplification of the pipeline logic, which can improve power consumption, area consumption, and maximum speed. The disadvantages are an increase in compiler complexity and a reduced ability to evolve the processor implementation independently of the instruction set (newer generations of chip might have a different pipeline depth and hence a different optimal number of delay cycles).

15.7 a.

	Cycle																
Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
lw \$1,0(\$2)	F	D	X	M	W												
addi \$1,\$1,1		F	D	D	X	M	W										
sw \$1,0(\$2)			F	F	D	X	M	W									
addi \$2,\$2,4					F	D	X	M	W								
sub \$4,\$3,\$2						F	D	X	M	W							
bne \$4,\$0,...							F	D	D								
lw \$1,0(\$2)										F	D	X	M	W			

b.

	Cycle																
Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
lw \$1,0(\$2)	F	D	X	M	W												
addi \$2,\$2,4		F	D	X	M	W											
addi \$1,\$1,1			F	D	X	M	W										
sw \$1,-4(\$2)				F	D	X	M	W									
sub \$4,\$3,\$2					F	D	X	M	W								
bne \$4,\$0,...						F	D										
lw \$1,0(\$2)								F	D	X	M	W					

CHAPTER 16 INSTRUCTION-LEVEL PARALLELISM AND SUPERSCALAR PROCESSORS

16.1 The following chart shows the execution of the given instruction sequence cycle by cycle. The stages of instruction execution are annotated as follows:

- IF Instruction fetch
- D Decode and issue
- EX1 Execute in LOAD/STORE unit
- EX2 Execute in ADD/SUB unit
- EX3 Execute in MUL/DIV unit
- WB Write back into register file and reservation stations

Instr ↓	cycle no.⇒	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
load	IF	D	EX1	EX1	WB												
load	IF	•	•	•	D	EX1	EX1	WB									
mul	IF	D	□	□	□	□	□	□	EX3	EX3	WB						
sub	IF	D	□	□	□	□	□	□	EX2	WB							
div	IF	•	•	•	•	•	•	•	•	•	D	□	EX3	EX3	EX3	EX3	WB
add	IF	•	•	•	•	•	•	•	•	D	□	EX2	WB				

Cycles in which an instruction is waiting for a reservation station are marked as • and the cycles in which an instruction is waiting for one or more operands are marked as □. As seen in the time chart, the issue and write back cycles for various instructions are as follows.

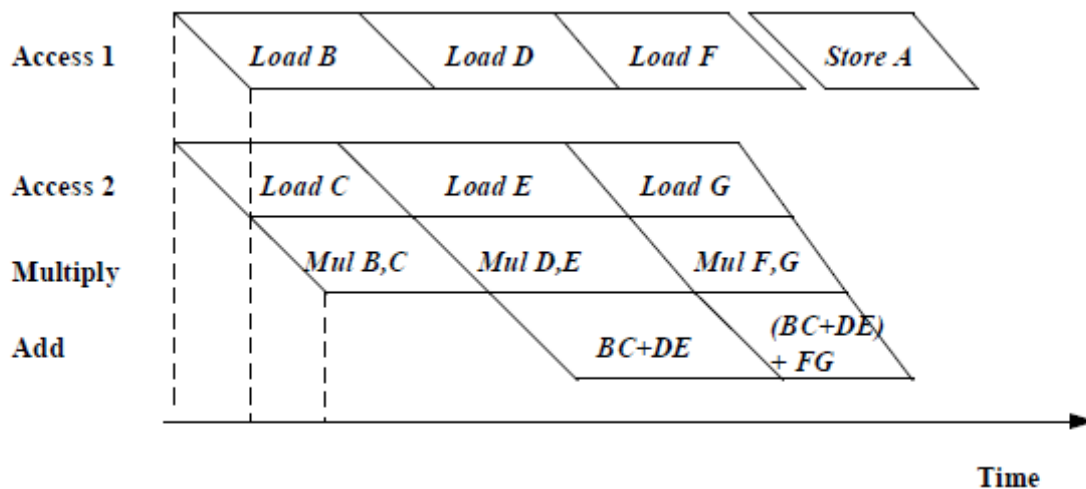
Instruction	issue cycle	write back cycle
load	1	4
load	4	7
mul	1	10
sub	1	9
div	10	16
add	9	12

- 16.2 a.** False. The reason is Scoreboarding does not eliminate either of WAR or WAW dependency.
- b.** False. There is a very swift law of diminishing returns for both number of bits and number of entries in a non correlating predictor. Additional resources only marginally improve performance.
- 16.3** Both (3) and (5) qualify but only (3) is strongly associated with super scalar processing. The reason being multiported caches would allow parallel memory access and thereby super scalar processing.
- 16.4 a.** 1. An instruction must wait for its dest operand to be completed before issue. (Otherwise we might have a problem with a load followed by an ALU to the same destination).
2. Any integer ALU instruction immediately following a branch must stall at least one cycle before issuing (after that the branch will flush the instr before it overwrites data).
- b.** We get to eliminate both the extra rules given in part (a). Eliminating the bubble between branches and ALU instrs will probably help a lot. Eliminating output-deps probably won't help matter so much (output deps that aren't also a true-dep followed by an antidep are rare, and in-order issue enforces both true-deps and anti-deps.) The real second improvement might come because the issue logic is simpler, thus less likely to impact the critical path.

CHAPTER 17 PARALLEL PROCESSING

17.1 Vectorization is a subset of MIMD parallel processing so the parallelization algorithms that detect vectorization all detect code that can be multiprocessed. However, there are several complicating factors: the synchronization is implicit in vectorization whereas it must normally be explicit with multiprocessing and it is frequently time consuming. Thus, the crossover vector length where the compiler will switch from scalar to vector processing will normally be greater on the MIMD machine. There are some vector operations that are more difficult on a MIMD machine, so some of the code generation techniques may be different.

17.2



- 17.3 a.** It reads the location, causing a state transition to S. Then it writes it, changing the state to M. So it ends up in state M.
- b.** P2 reads it, so P2 has it in state S (in the top variation, it goes to state I in P1, while in the bottom it goes to state S in P1). Now P2 writes it, so it goes to state M in P2 and state I in P1.

17.4 The problem is that CPU A cannot prevent CPU B from writing to location x between the time CPU A reads location x and the time CPU A writes location x. For fetch and increment to be atomic, CPU A must be able to read and write x without any other processor writing x in between. For example, consider the following sequence of events and corresponding state transitions and operations:

Event	CPU A	CPU B
1	Read(x); I->CS; send CR	
2		Snoop CR; CE->CS
3		Write(x); CS->OE; send CI
4	Snoop CI; CS->I	
5	Write(x); I->OE; send CRI	
6		Snoop CRI; OE->I

Note that it is OK for CPU B to read x in between the time CPU A reads x and then writes x because this case is the same as if CPU B read x before CPU A did the fetch and increment. Therefore the fetch and increment still appears atomic

CHAPTER 18 MULTICORE COMPUTERS

- 18.1 a.** The code for each thread would look very similar to the assembly code above, except the array indices need to be incremented by N instead of 1, and each thread would have a different starting offset when loading/storing from/to arrays A, B, and C
- b.** 4
- c.** Yes, we can hide the latency of the floating-point instructions by moving the add instructions in between floating point and store instructions – we'd only need 3 threads. Moving the third load up to follow the second load would further reduce thread requirement to only 2.
- d.** $8/48=0.17\text{flops/cycle}$.

CHAPTER 19 CONTROL UNIT OPERATION

19.1 a. Addresses Contents

0x0000B128 0x0200EC00

0x0000B12C 0x0300EC04

0x0000B130 0x0100EC08

(1st byte: opcode (e.g., 0x02), remaining 3 bytes are address of data)

.....

0x0000EC00 0x00000016 ; (a=22=0x16)

0x0000EC04 0x0000009E ; (b=158=0x9E)

0x0000EC08 0x00000000 ; (c=0=0x00, or it can be anything)

b. Instruction Contents

PC → MAR 0x0000B128

M → MBR 0x0200EC00

MBR → IR 0x0200EC00

IR → MAR 0x0000EC00

M → MBR 0x00000016

MBR → AC 0x00000016

PC → MAR 0x0000B12C

M → MBR 0x0300EC04

MBR → IR 0x0300EC04

IR → MAR 0x0000EC04

M → MBR 0x0000009E

MBR + AC → AC 0x00000B4

PC → MAR 0x0000B130

M → MBR 0x0100EC08

MBR → IR 0x0100EC08

IR → MAR 0x0000EC08

AC → MBR 0x000000B4

MBR → M 0x000000B4

APPENDIX B ASSEMBLY LANGUAGE AND RELATED TOPICS

B.1

n	\$a0
r	\$v0
r*r	\$t0

```
sqrt: addi $v0, $zero, 0 # r := 0
loop: mul  $t0, $v0, $v0 # $t0 := r*r
      bgt  $t0, $a0, end  # if (r*r > n) goto end
      addi $v0, $v0, 1    # r := r + 1
      jloop              # goto loop
end:   addi $v0, $v0, -1   # r := r - 1
      jr   $ra           # return r
```

Note that since we don't use any of the \$s registers, and we don't use jal, and since the problem said not to worry about the frame pointer, we don't have to save/restore any registers on the stack. We use \$a0 for the argument n, and \$v0 for the return value because these choices are mandated by the standard MIPS calling conventions. To save registers, we also use \$v0 for the program variable r within the body of the procedure.

B.2

```
# Local variables:
# b$a0 Base value to raise to some power.
# e$a1 Exponent remaining to raise the base to.
# r$v0 Accumulated result so far.

intpower: addi $v0, $zero, 1 # r := 0.
while:    blez $a1, endwh    # Break out of while
      loop when e<=0.
      mul    $v0, $v0, $a0  # r := r * b.
      addi   $a1, $a1, -1   # b := b - 1.
      b while              # Continue looping.
endwh:    jr   $ra          # Return result r already in $v0.
```

B.3

```
AND    $r4, $r2, $r3
SLL    $r4, $r4, 1
XOR    $r4, $r4, $r2
XOR    $r4, $r4, $r3
```