STRUCTURED COMPUTER ORGANIZATION

FIFTH EDITION

PROBLEM SOLUTIONS

ANDREW S. TANENBAUM
Vrije Universiteit
Amsterdam, The Netherlands

PRENTICE HALL

UPPER SADDLE RIVER, NJ 07458

SOLUTIONS TO CHAPTER 1 PROBLEMS

1. a. A translator converts programs in one language to another.
b. An interpreter carries out a program instruction by instruction.
c. A virtual machine is a conceptual machine, one that does not exist.

2. An interpreter executes a program by fetching the first instruction, carrying it out, then fetching the next one, and so on. A translator first converts the original program into an equivalent one in another language and then runs the new program.

3. It is possible, but there are problems. One difficulty is the large amount of code produced. Since one ISA instruction does the work of many microinstructions, the resulting program will be much bigger. Another problem is that the compiler will have to deal with a more primitive output language, hence it, itself, will become more complex. Also, on many machines, the microprogram is in ROM. Making it user-changeable would require putting it in RAM, which is much slower than ROM. On the positive side, the resulting program might well be much faster, since the overhead of one level of interpretation would be eliminated.

4. During the detailed design of a new computer, the device and digital logic levels of the new machine may well be simulated on an old machine, which puts them around level 5 or 6.

5. Each level of interpretation slows down the machine by a factor of n/m. Thus the execution times for levels 2, 3, and 4 are kn/m, kn^2/m^2, and kn^3/m^3, respectively.

6. Each additional level of interpretation costs something in time. If it is not needed, it should be avoided.

7. You lose a factor of n at each level, so instruction execution times at levels 2, 3, and 4 are kn, kn^2, and kn^3, respectively.

8. Hardware and software are functionally equivalent. Any function done by one can, in principle, be done by the other. They are not equivalent in the sense that to make the machine really run, the bottom level must be hardware, not software. They also differ in performance.

9. Not at all. If you wanted to change the program the difference engine ran, you had to throw the whole computer out and build a new one. A modern computer does not have to be replaced because you want to change the program. It can read many programs from many CD-ROMs.

10. A typical example is a program that computes the inner product of two arrays, *A* and *B*. The first two instructions might fetch *A*[0] and *B*[0], respectively. At the end of the iteration, these instructions could be incremented to point to *A*[1] and *B*[1], respectively. Before indexing and indirect addressing were invented, this was done.

11. Raw cycle time is not the only factor. The number of bytes fetched per cycle is also a major factor, this increasing with the larger models. Memory speed and wait states play a role, as does the presence of caching. A better I/O architecture causes fewer cycles to be stolen, and so on.

12. The design of Figure 1-5 does I/O one character at a time by explicit program command. The design of Figure 1-6 can use DMA to have the controller do all the work, relieving the CPU of the burden, and thus making it potentially better.

13. Each person consumes 730 tags per nonleap year. Multiply by 300 million and you get 219 billion tags a year. At a penny a tag, they cost $2.19 billion dollars a year. With GDP exceeding $10 trillion, the tags add up to 0.02% of GDP, not a huge obstacle.

14. The following appliances are normally controlled by embedded systems these days: alarm-clock radios, microwave ovens, television sets, cordless telephones, washing machines, sewing machines, and burglar alarms.

15. According to Moore's law, next year the same chip will have 1.6 times as many transistors. This means that the area of each transistor will be 1/1.6 or 0.625 times the size of this year's transistors. Since the area goes like the square of the diameter, the diameter of next year's transistors must be 0.079 microns.

SOLUTIONS TO CHAPTER 2 PROBLEMS

1. The data path cycle is 20 nsec. The maximum number of data path cycles/sec is thus 50 million. The best the machine could do is thus 50 MIPS.

2. The program counter must be incremented to point to the next instruction. If this step were omitted, the computer would execute the initial instruction forever.

3. You cannot say anything for sure. If computer 1 has a five-stage pipeline, it can issue up to 500 million instructions/second. If computer 2 is not pipelined, it cannot do any better than 200 million instructions/sec. Thus without more information, you cannot say which is faster.

4. On-chip memory does not affect the first three principles. Having only LOADs and STOREs touch memory is no longer required. There is no particular reason not to have a memory-to-memory architecture if memory references are as fast as register references. Likewise, the need for many registers becomes less in this environment.

5. A pipeline processor is better. Array processors are useful only if the problem contains inherent parallelism.

6. The monastery resembles Figure 2-7, with one master and many slaves.

7. The access time for registers is a few nanoseconds. For optical disk it is a few hundred milliseconds. The ratio here is about 10^8.

8. Sixty-four 6-bit numbers exist, so 4 trits are needed. In general, the number of trits, k, needed to hold n bits is the smallest value of k such that $3^k \geq 2^n$.

9. A pixel requires 6 + 6 + 6 = 18 bits, so a single visual frame is 1.8×10^7 bits. With 10 frames a second, the gross data rate is 180 Mbps. Unfortunately, the brain's processing rate is many orders of magnitude less than this. As an experiment, try watching the random noise on a color television when no station is broadcasting and see if you can memorize the color bit pattern in the noise for a few minutes.

10. With 44,000 samples per second of 16 bits each, we have a data rate of 704 kbps.

11. There are 2 bits per nucleotide, so the information capacity of the human genome is about 6 gigabits. Dividing this number by 30,000, we get about 200,000 bits per gene. Just think of a gene as an 25-KB ROM. This estimate is an upper bound, because many of the nucleotides are used for purposes other than coding genes.

12. For efficiency with binary computers, it is best to have the number of cells be a power of 2. Since 268,435,456 is 2^{28}, it is reasonable, whereas 250,000,000 is not.

13. From 0 to 9 the codes are: 0000000, 1101001, 0101010, 1000011, 1001100, 0100101, 1100110, 0001111, 1110000, and 0011001.

14. Just add a parity bit: 00000, 00011, 00101, 00110, 01001, 01010, 01100, 01111, 10001, and 10010.

15. If the total length is $2^n - 1$ bits, there are n check bits. Consequently, the percentage of wasted bits is $n/(2^n - 1) \times 100\%$. Numerically for n from 3 to 10 we get: 42.9%, 26.7%, 16.1%, 9.5%, 5.5%, 3.1%, 1.8%, and 1.0%.

16. With 4096 bits/sector and 1024 sectors/track, each track holds 4,194,304 bits At 7200 RPM, each rotation takes 1/120 sec. In 1 sec it can read 120 tracks for a rate of 503,316,480 bits/sec or 62,914,560 bytes/sec.

17. At 160 Mbytes/sec and 4 bytes/word, the disk transfer rate is 40 million words/sec. Of the 200 million bus cycles/sec, the disk takes 1/5 of them. Thus the CPU will be slowed down by 20 percent.

18. Logically it does not matter, but the performance is better if you allocate from the outside in. One rotation of the outermost track takes as long as one rotation of the innermost track (because hard disks rotate with constant angular velocity), but there are more sectors on the outermost track, so the transfer rate is higher. It is smarter to use the high-performance sectors first. Maybe the disk will never fill up and you will never have to use the lowest-performance sectors.

19. A cylinder can be read in four rotations. During the fifth rotation, a seek is done to the next cylinder. Because the track-to-track seek time is less than the rotation time, the program must wait until sector zero comes around again. Therefore, it takes five full rotations to read a cylinder and be positioned to start reading the next one. Reading the first 9999 cylinders thus takes 49,995 rotations. Reading the last cylinder requires four rotations, because no final seek is needed. The 49,999 rotations at 10 msec/rotation take 499.99 sec. If the sectors are skewed, however, it may be possible to avoid the fifth rotation per track.

20. RAID level 2 can not only recover from crashed drives, but also from undetected transient errors. If one drive delivers a single bad bit, RAID level 2 will correct this, but RAID level 3 will not.

21. In mode 2, the data streams at 175,200 bytes/sec. In a 80-min time span, the number of seconds is 5920, so the size of a 80-min mode 2 CD-ROM is 840,960,000 bytes or 802 MB. Of course, in mode 2 there is no error correction, which is fine for music but not for data. In mode 1, only 2048/2336 of the bits are available for data, reducing the payload to 737,280,000 bytes or 703 MB.

22. The mode does not matter, since the laser has to pulse for preamble bits, data bits, ECC bits, and all the overhead bits as well. The gross data rate at 1x is 75 sectors/sec, each sector consisting of $98 \times 588 = 57,624$ bits. Thus 4,321,800 bits/sec fly by the head at 1x. At 10x, this is 43,218,000 bits/sec. Thus each pulse must last no more than 23.14 nsec (actually slightly less, since there is a blank interval between pulses).

23. Each frame contains 345,600 pixels or 1,036,800 bytes of information. At 30 fps, the rate per second is 31,104,000 bytes/sec. In 133 minutes this amounts to 2.482×10^{11} bytes. The disk capacity is 3.5×2^{30} which is about

3.758×10^9 bytes. We are off by a factor of 66, so the compression has to be 66x.

24. The read time is the size divided by the speed: $25 \times 2^{30} / 4.5 \times 10^{20}$ or about 5688.9 sec. This is almost an hour and 35 minutes.

25. The CPU wastes millions of instructions while waiting for mechanical I/O (e.g., a printer) to finish. The problem can be handled using DMA and causing an interrupt after a block has been transferred. In some cases, the CPU can copy a block of data to the device, which then handles it on its own and causes an interrupt when it is ready for more.

26. The usual way to handle this would be to have a bank of 256 24-bit mapping registers in the hardware. Whenever a byte was fetched from the video RAM, the 8-bit number would be used as an index into the mapping registers. The register selected would deliver the 24 bits to drive the display (typically 8 bits each for the red, green, and blue electron guns). Thus indeed 2^{24} colors are available, but at any instant, only 256 are available. Changing colors means reloading the mapping registers.

27. The display must paint $1600 \times 1200 \times 75$ pixels/sec. This is a total of 144,000,000 pixels Thus the pixel time is 6.944 nsec.

28. A page has 4000 characters. Each character uses 25% of 4 mm^2, or 1 mm^2. Thus a page has 4000 mm^2 of toner. With a thickness of 25 microns (0.025 mm), the volume of toner on a page is 100 mm^3. The capacity of the toner cartridge is 400 cm^3 or 400,000 mm^3. A cartridge is good for 4000 pages.

29. At any speed above 110 bps, there is one start bit, one stop bit, and one parity bit, so 7 of the 10 bits are data. Thus 70 percent of the bits are data. Since $0.7 \times 5600 = 39,200$, the true data rate is 39,200 bps.

30. Each interval can transmit 6 bits, so the data rate is 6*n* bps.

31. A 12 MHz cable with QAM-64 has a data rate of 72 Mbps. With *nf* computers sharing the bandwidth, each user gets $72/nf$-Mbps. Thus the cable user gets better service if $72/nf > 2$. An alternative way to write this is $nf < 36$. In other words, if the 72 Mbps bandwidth is being shared by 36 active users, it is the same as 2-Mbps ADSL; with fewer users, cable wins; with more users, ADSL wins.

32. Each uncompressed image file is 18 million bytes. After 5x compression, it is 3.6 million bytes. To write this in 2 sec requires a data rate of 1.8 MB/sec.

33. The uncompressed image is 48 million bytes. The compressed image is 9.6 million bytes. The number of images stored is thus $2^{30}/9.6$ million or 111 with a few megabytes left over.

34. A typical computer science textbook has about a million characters, so it needs about 1 MB. Ten thousand books require 10^{10} bytes. A CD-ROM holds 650 MB, so you need 16 CD-ROMs. A dual-layer, double-sided DVD holds 17 GB, so the whole library fits on one DVD.

SOLUTIONS TO CHAPTER 3 PROBLEMS

1. The order is

hamburger or hot dog and french fries

which can be parsed as

hamburger or (hot dog and french fries)

or as

(hamburger or hot dog) and french fries

The first parse leads to alternatives a and d. The second parse leads to alternatives e and d. Thus a, d, and e are all possible. Choice h is ruled out on educational grounds: the cook is too stupid to get the joke.

2. He should point to one of the roads and ask: "If I were to ask the other gang if this is the road to Disneyland, what would they say?" If the answer is no, he should take the road; if the answer is yes, he should not take it. The validity of this solution can easily be seen by trying all four combinations of right/wrong road and liars/truth-tellers.

3. The truth table required is as follows:

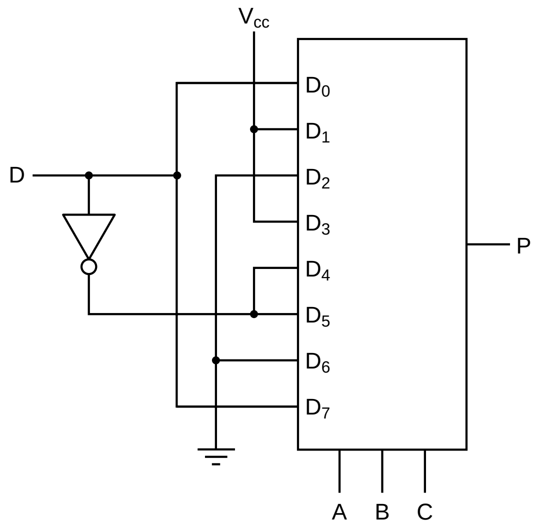
P	Q	P AND Q	P AND NOT Q	(P AND Q) OR (P AND NOT Q)
0	0	0	0	0
0	1	0	0	0
1	0	0	1	1
1	1	1	0	1

4. With three variables, the truth table has eight rows, so a function can be described by an 8-bit number. Thus 256 functions exist. With n variables, the truth table has $k = 2^n$ rows and there are 2^k functions.

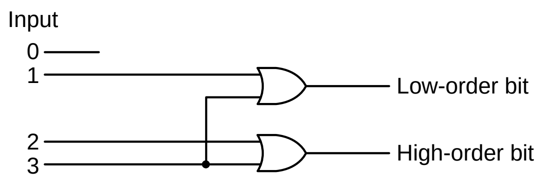
5. Call the two variables A and B. Connect the inputs to the first NAND gate to A and B. Take the output and feed it into both inputs of the second NAND gate and presto, AND.

6. Inputs D_1, D_2, D_4, and D_7 are all connected to ground The other four inputs are tied to V_{cc}.

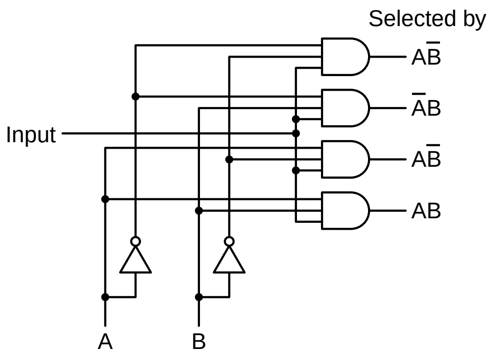
7. Input line D_0 supplies the output for truth table rows 0000 and 0001. Input line D_1 supplies the output for truth table rows 0010 and 0011, and so on. For each case, the function values for the two rows can be 00, 01, 10, and 11. If they are 00, just wire the input to ground; if they are 11, just wire it to V_{cc}. If they are 01, notice that they are just the same as the fourth input variable, D, so wire it to D. If they are 10, wire it to $\overline{D}$. The truth table values for the example, from 0000 to 1111, are 0111001110100001, so the circuit is as follows.



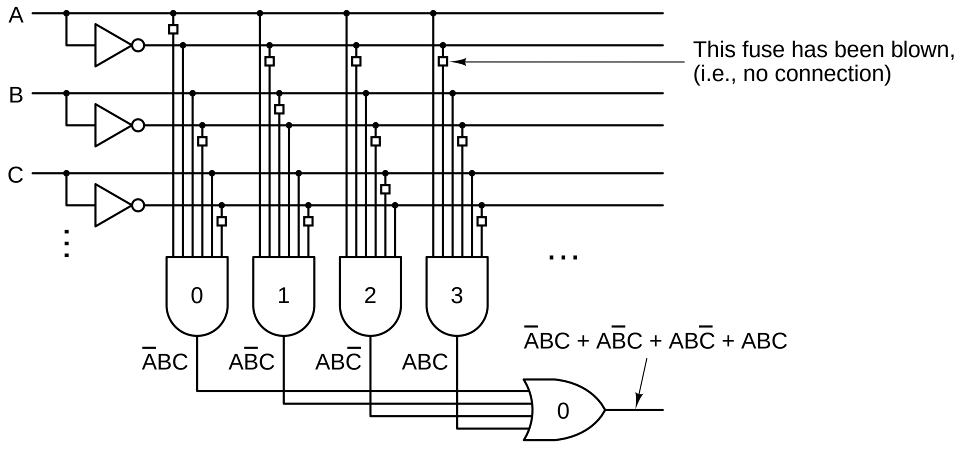
8. The encoder looks like this. Note that line 0 is not used.



9. The demultiplexer looks like this.



10. The relevant part of the PLA is as follows.


A
B
C
This fuse has been blown, (i.e., no connection)
0
1
2
3
$\overline{A}BC$
$A\overline{B}C$
$AB\overline{C}$
ABC
$\overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$
0

11. It is a half adder with C as the sum and D as the carry.

12. The chip needs four pins for the first operand, four pins for the second operand, four pins for the result, one pin for carry in, and one pin for carry out (to make it cascadable), plus power and ground, for a total of 16 pins.

13. The carry into stage i can be written as $C_i = P_{i-1} + S_{i-1}C_{i-1}$, where P_{i-1} is the product term $A_{i-1}B_{i-1}$ and S_{i-1} is the sum term $A_{i-1} + B_{i-1}$. This result follows directly from the fact that a carry is generated from a stage if both operands are 1, or if one operand and the carry in are both 1. For example,

$$C_0 = 0$$

$$C_1 = P_0 + S_0C_0 = P_0$$

$$C_2 = P_1 + S_1C_1 = P_1 + P_0S_1$$

$$C_3 = P_2 + S_2C_2 = P_2 + P_1S_2 + P_0S_1S_2$$

$C_4 = P_3 + S_3C_3 = P_3 + P_2S_3 + P_1S_2S_3 + P_0S_1S_2S_3$

As soon as the inputs, *A* and *B*, are available, all the *P* and *S* terms can be generated simultaneously in one gate delay time. Then the various AND terms such as $P_0S_1S_2$ can be generated in a second gate delay. Finally, all the carries can be produced in a third gate delay. Thus all the carries are available after three gate delays, no matter how many stages the adder has. The price paid for this speedup is a considerable number of additional gates, of course.

14. The timing of the circuit can be found by writing a 0 on each of the input lines and then tracing them through the circuit, adding 1 at each gate. The *A* input takes 2 nsec to become available, so the output of the logic unit takes 4 nsec and the final output for a Boolean operation takes 5 nsec. The decode lines that drive the logic unit each have two gate delays, so the enable lines are set in plenty of time. The adder also takes 3 nsec to produce its contribution to the output gate after it has *A*. Worst case through the whole circuit is 6 nsec.

15. The ALU is already capable of subtraction. Note that in 2's complement, $B - A = B + (-A)$. To get $-A$, we can use $\overline{A} + 1$. Thus to subtract *A* from *B* we need to add B, $\overline{A}$ and then increment the sum. The circuit can already do this by asserting INVA and INC and then adding the two inputs.

16. A basic cycle is 11 nsec, including propagation. Sixteen cycles take 176 nsec. However, the last propagation is not needed, so the answer is available 175 nsec after the start.

17. One way is to negate ENB, forcing *B* to 0, then choosing function code 01 to select $\overline{B}$ as the ALU function. The 1's complement of 0 is −1 in 2's complement (all 1 bits). As long as INC is negated, the control lines for *A* do not matter. A second way is to negate and invert *A*, putting all 1s on the *A* input. Then negate ENB to force *B* to 0. With *A* equal to −1 and *B* equal to 0, we can either add or OR them together.

18. The NAND latch is wired the same way as the NOR latch. Normally, both inputs should be 1 to achieve consistency between input and output.

19. Use the same circuit, but replace the AND gate in the pulse generator by a NOR gate. The only time both inputs will be low is just after the falling edge.

20. The design uses two AND gates for chip enable logic plus two AND gates per word select line plus one AND gate per data bit. For a 256×8 memory this comes to 2 + 512 + 2048 = 2562 AND gates. The circuit also uses one OR gate for each bit in the word; hence eight of them would be needed.

21. It will not fly. Each of the four flip-flops needs three pins, for *D*, *Q*, and $\overline{Q}$. In addition, the chip needs clock, power, and ground. Unless you plan on putting out the world's first 15-pin chip, you are going to have to use a 16-pin package, thus wasting a pin. Thus the design is not very efficient. Since an extra pin is available, you might as well do something with it to make the chip more flexible. For example, you might use the sixteenth pin to reset all the flip-flops.

22. The pins can be multiplexed in time. For example, with $n/2$ pins, half the bits are presented in one cycle, and the other half in a succeeding cycle. Many dynamic RAMs work this way. Even more extreme is to feed the address serially into the chip a bit at a time using a single pin.

23. A 32-bit-wide data bus means that 32 chips must be used in parallel, each chip providing 1 bit. Thus the smallest memory consists of 32 chips, which is 32 megabits or 4 Mbytes.

24. With a 20-nsec clock period, $\overline{\text{MREQ}}$ might be asserted as late as 13 nsec into T_1. The data are required 2 nsec before the high-to-low transition in T_3, which occurs 10 nsec after the start of the cycle. From the midpoint of T_1 to the midpoint of T_3 is 40 nsec. Since the memory cannot start until 3 nsec after the transition in the first cycle and has to be done 2 nsec before the transition in the third cycle, in the worst case the memory has only 35 nsec in which to respond.

25. The memory would now have $20 - 3 - 4 = 13$ nsec to respond after $\overline{\text{MREQ}}$ and $\overline{\text{RD}}$ are asserted. There is not a lot of margin left, but if all the memory chips can always respond in 10 nsec, the system will still work.

26. In principle it could be negative, but it has to be asserted after the address is stable because on a write, the memory must not start changing the bytes addressed until the address is valid. On reads it does not matter.

27. In regular mode, it takes three cycles per transfer. In block mode, it takes one cycle per transfer once the transfer has started. Therefore block transfers have almost three times as much bandwidth. The bus width does not matter, so it is still three times more, even for 32-bit transfers.

28. The CPU cannot assert $\overline{\text{MSYN}}$ until it has provided an address and told what it wants, so we have

$T_{A1} < T_{MSYN1}$,
$T_{MREQ1} < T_{MSYN1}$, and
$T_{RD1} < T_{MSYN1}$.

The data cannot be asserted before $\overline{\text{MSYN}}$, so

$T_{MSYN1} < T_{D1}$ and
$T_{D1} < T_{SSYN1}$.

Once $\overline{\text{SSYN}}$ has been asserted, the master can clean up, so

$T_{SSYN1} < T_{MSYN2}$,
$T_{SSYN1} < T_{RD2}$,
$T_{SSYN1} < T_{MREQ2}$, and
$T_{SSYN1} < T_{A2}$.

Finally,

$T_{MSYN2} < T_{D2}$ and
$T_{MSYN2} < T_{SSYN2}$.

29. Yes there is. When reading the lower half of a word, the lower half of the 32 data lines is used. However, when reading the upper half, the bus designers have a choice of using the upper half of the lines or shifting everything downward to the lower half. The former is an unjustified bus; the latter is a justified bus. Both types are in use.

30. The interrupt acknowledge cycle is needed so that the interrupting device (or the interrupt controller) can see that the interrupt has been accepted and the CPU wants the interrupt vector number. Since the CPU can disable interrupts for an arbitrarily long time, after asserting an interrupt, a device might have to wait a long time before the interrupt will be accepted, so it needs some way to see on which cycle it must output the interrupt vector number.

31. At 200 MHz, a cycle is 5 nsec. Four cycles take 20 nsec. A word is 8 bytes, so the machine needs 8 bytes every 20 nsec, for a data rate of 400 Mbytes/sec.

32. There are seven combinations: one for words, two for half-words (upper and lower), and four for bytes. Three pins are enough for seven combinations if complete encoding is used. This minimizes pins. On the other hand, it requires an external decoder, so there is something to be said for having four lines, one per byte, saying whether that byte is needed or not.

33. The Pentium 4 is capable of addressing 2^{36} bytes of memory. If they were addressed a word at time, there would be 2^{34} possible words to address. It would take a 34-bit address to address them all. However, the Pentium 4 has only 33 address lines. In theory, it could ask for a 64-bit word in two consecutive 32-bit cycles, but this change would mean redesigning the processor.

34. The average access time is $0.20 \times 1 + 0.60 \times 2 + 0.20 \times 10 = 3.4$ nsec.

35. Not very likely. The 8255A provides 24 I/O lines, but the 8051 already has 32 such lines built in. Only if more than 16 are needed would an 8255A make sense.

36. Each frame contains 921,600 bytes. In one second, 30 such frames are needed, or 27,648,000 bytes. If the data travel over the bus twice, the needed bandwidth is 55,296,000 bytes/sec or 52.734 MB/sec (1 MB is 2^{20} bytes.)

37. The FRAME# signal starts a PCI bus transaction. The Pentium 4 signal that says that it is ready to go is ADS#, so this should connect to FRAME#.

38. The DEVSEL# signal is not really essential. It is nice to know that somebody out there wants you, but the real information is conveyed by TRDY#.

39. For 8x operation, 8 lanes are needed each way for a gross capacity of 40 Gbps. The net capacity is then 32 Gbps.

40. With a 32-bit bus, the disk must make 262,144 transfers in 5 msec (one rotation) or about 52,429 transfers/msec. In this time, it uses up 52,429 of the 100,000 bus cycles available per millisecond, leaving only 47,571 for the CPU. The CPU is therefore slowed down to 47.571 percent of its full speed.

41. The frame rate is 1000 frames/sec. At 64 bytes/frame, the maximum data rate for a device is 1,023,000 bytes/sec or 999.023 MB/sec.

42. As the circuit now stands, the PIO is selected by all addresses beginning with 11, that is, any address above 48K. Adding a third line causes it to be selected only by addresses above 56K.

SOLUTIONS TO CHAPTER 4 PROBLEMS

1. Only one register may be gated out onto the B bus, so a single 4-bit field is sufficient to specify which one. The C bus may have many registers selected to be loaded, so a full bit map is required.

2. The Boolean function that is needed is

```
F = (JAMZ AND Z) OR (JAMN AND N) OR HIBIT
```

where HIBIT is the high-order bit of the NEXT_ADDRESS field. This can be implemented with two AND gates that feed an OR gate.

3. No. No matter what is in *MBR*, the next microinstruction will come from 0x1FF. The contents of *MBR* are completely irrelevant. There is no point ORing *MBR* with 0x1FF since the result will always be 0x1FF, no matter what is in *MBR*. The OR does nothing.

4. All compilers will add

```
BIPUSH 5
ISTORE k
```

at the label *L2*. However, an optimizing compiler will notice that *i* is never used after the if statement. Therefore the fourth and fifth instructions (*ISTORE* and *ILOAD*) are not needed. The code becomes

```
        ILOAD j
        ILOAD k
        IADD
        BIPUSH 3
        I_CMPEQ L1
        ILOAD j
        BIPUSH 1
        ISUB
        ISTORE j
        GOTO L2
L1:     BIPUSH 0
        ISTORE k
L2:     BIPUSH 0
        ISTORE i
```

5. Here are two ways to do it:

```
ILOAD k                 ILOAD k
ILOAD n                 ILOAD n
IADD                    BIPUSH 5
BIPUSH 5                IADD
IADD                    IADD
ISTORE i                ISTORE i
```

Here we have used the associative law, that is the fact that $(j + k) + 4$ is the same as $j + (k + 4)$, at least if the possibility of overflow is ignored.

6. The statement is something like i = (j – k – 6) + (j – n – 7).

7. The text is correct. If the condition is true, the *Z* bit is 1 and control passes to *L1*. The address of *L1* is computed by ORing the *Z* bit (which is a 1) into *MPC*. This means the *L1* is in the top half of the control store. Consequently, *L2* must be in the bottom half.

8. It is not possible to subtract *TOS* from *MDR* as one of the operands must be *H* or a small constant. The only way to compute *MDR* – *TOS* is to first copy *MDR* to *H*.

9. The code and the number of microinstructions for each one is *ILOAD* (6), *ILOAD* (6), *IADD* (4), *ISTORE* (7). This adds up to 23 microinstructions. At 2.5 GHz, each microinstruction takes 0.4 nsec, so 23 of them take 9.2 nsec.

10. The code and the number of microinstructions for each one is *ILOAD* (3), *ILOAD* (3), *IADD* (3), *ISTORE* (5). This adds up to 14 microinstructions. At 2.5 GHz, each microinstruction takes 0.4 nsec, so 14 of them take 5.6 nsec.

Based on this result and the one of the previous problem, it appears that Mic-2 execution times are 5.6/9.2 of Mic-1 execution times. Therefore, a 100-sec Mic-1 program should take 60.87 sec.

11. Something like this would do the job:

```
pop2a   SP = SP – 1
pop2b   MAR = SP = SP – 1; rd
pop2c
pop2d   TOS = MDR; goto Main1
```

12. Currently, local 0 is the link pointer, which is rarely used. Thus having cheap access to it is not especially useful. For the full JVM machine, it would be better to have *LV* point to the first local variable (the first parameter), with the link pointer at offset −1 from it.

13. a. The arithmetic result is $floor(value)/2^s$, where s is the unsigned 5-bit integer at the top of stack, and value is the second word on the stack, interpreted as a 32-bit signed integer. Note that for negative integers, this is not truncated division by a power of two. It is more complicated than other instructions because it must include a do loop for a variable shift count.

b. This sequence uses the shift control signal sra1, not otherwise used in the Mic-1.

14. a. The arithmetic result is $value \times 2^s$, where s is the unsigned 5-bit integer at the top of stack, and *value* is the second word on the stack, interpreted as a 32-bit signed integer. However, the result may not be representable in 32 bits.

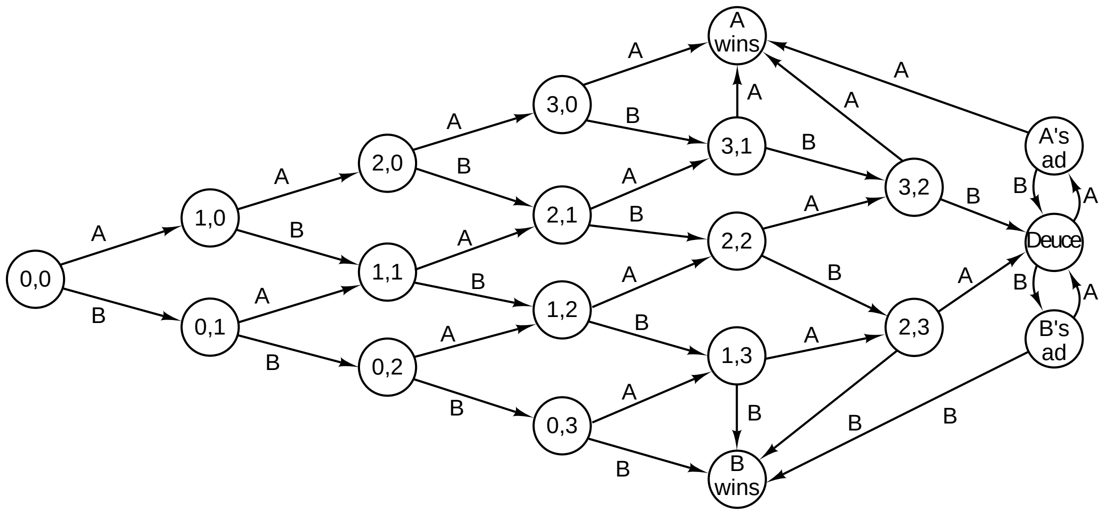
b. The microcode sequence must use repetitive addition to achieve the left arithmetic shift operation. It is more complicated than other instructions because it must include a do loop for a variable shift count.

15. It needs to find the *OBJREF* word, so it has to know how many parameters to skip over to locate it down in the stack.

16. One possible implementation is

```
dload1   MAR = LV + MBR1U; rd
dload2   H = MAR + 1
dload3   MAR = SP = SP + 1; wr
dload4   MAR = H; rd
dload5   MAR = SP = SP + 1; wr
dload6   TOS = MDR; goto (MBR)
```

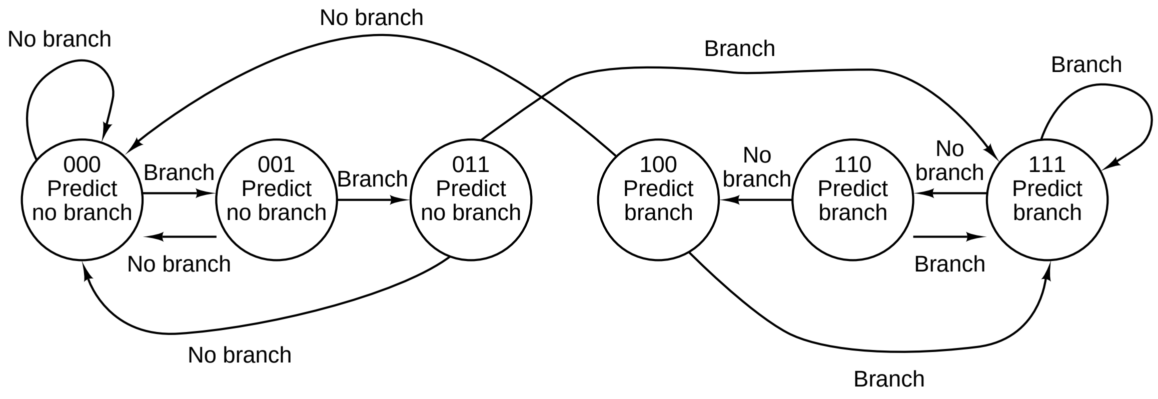
17. The finite-state machine for tennis looks like this.



Deuce means that the score is tied and that if either player scores two consecutive points he wins. *X's* ad(vantage) means that *X* needs one point to win.

18. The states (2, 2) and Deuce are equivalent and can be merged. Remove (2, 2) and have the arcs leading into it go to Deuce instead. Likewise, (3, 2) is really *A*'s ad and (2, 3) is really *B's* ad.

19. The FSM looks like this. The state labels are not unique.



20. A 5-byte version would work fine. A 4-byte version would not work because if there were only 1 byte left in the queue and *MBR* 2 were referenced, the reference could not be satisfied, but there would be no room to fetch another word.

21. Definitely. We have been assuming a 100% cache hit rate so far. That is unrealistic. Sometimes a memory fetch will be a cache miss and take a long time. Having a large queue of previously fetched bytes will allow the machine to continue to run even when a memory reference takes a long time. Thus during much of the long memory fetch time the CPU may still be running. With a small shift register in the IFU it may have to stall much earlier.

22. Remember that due to the way branching works, the true addresses *must* be exactly 0x100 more than the false address. By always pairing *T* and *F*, it is easy to make sure they are at a distance of 0x100. On the other hand, all the conditional branches end up at *T* if taken and then go to *GOTO* 1, so if *GOTO* 1 were positioned at exactly 0x100 above *F*, it would have worked. Doing so would have saved 1 word of control store, but it would not have made the machine any faster.

23. The main problem is that the micro-operations have to be in sequence so they can be read out efficiently. The IJVM opcode for *POP* is 0x57, so its sequence would have to begin there. The trouble is, this sequence is three micro-operations long, so they would go in slots 0x57, 0x58, and 0x59. However, *DUP* must begin in 0x59, so there is a conflict. In addition, the scheme used in the Mic-4 allows the micro-operations to be packed densely. If there are *k* micro-operations in all, then only *k* table entries are needed. In a scheme like the one the Mic-1 has, at least 256 entries are needed.

24. The mean access time is $0.6 \times 5 + 0.35 \times 20 + 0.05 \times 80$. This gives 14 nsec.

25. Not really. If the cache is write allocate, then the line will be brought in at the time of the write, so subsequent reads are free. If the cache is not write allocate, the write will go through to memory, but the first read will bring the line in, and the rest of the reads are free. The only gain here is the saving of one write through to memory. The downside of write allocate is that if there are no subsequent reads, the line will have been brought in for nothing. Thus write allocate is a gamble that the line will be *written* again soon.

26. The reviewer was wrong. In an *n*-way associative cache, there is hardware to compare all the tags in parallel to the tag portion of the address being looked up. It does not matter if there are three, four, or 19 comparisons being done in parallel. Many commercial machines have 3-way associative caches.

27. A conditional branch ties up the fetch unit for 4 cycles—one fetch and three dead cycles after it. The mean number of cycles spent using the fetch unit is $0.8 \times 1 + 0.2 \times 4 = 1.6$ fetch cycles/instruction. Without this problem it would have been 1.0 fetch cycles per instruction. Thus the average instruction takes 60% more time and the machine runs at 1/1.6 or 5/8 of full speed.

28. Prefetching will only be on the right track if all four branches are correctly predicted. The chance that this happens is 0.9^4, which is about 0.66.

29. Here is what happens:
Cycle 6: I6 and I7 are issued
Cycle 7: I4 is retired
Cycle 8: I5, I6, and I7 are retired
Cycle 9: I8 is issued
Cycle 11: I8 is retired

30. A WAW dependence occurs when a register that is being written is overwritten before the first one is through. If there are no intervening reads of the register and no side effects, the first operation is wasted, so it could be removed, aborted early, or have its result overwritten.

SOLUTIONS TO CHAPTER 5 PROBLEMS

1. The value is 3×2^{24} or 50331648.

2. The Pentium 4 allows 1-byte instructions. Many of the most common instructions are, in fact, one byte. This leads to shorter, faster, programs.

3. It can be done by allocating the high-order 3 bits as the main opcode. Seven instructions use 000 to 110 in these bits, leaving 33 bits for the two addresses and the register. Opcode 111 uses the first 15-bit address field to distinguish the 500 instructions. With 32768 opcode slots available, there is plenty of room. The zero-address instructions are encoded with 111 in the main opcode and one of the 32768 - 500 = 32268 unused numbers in the first address field.

4. Each two-address instruction uses up 4096 bit combinations. With n of them, $4096n$ opcodes are needed. The number of opcodes remaining for the one-address instructions is then $65536 - 4096n$. Each one-address instruction uses 64 slots, so at most $(65536 - 4096n)/64 = 1024 - 64n$ are available.

5. No. The three requirements take $4 \times 8 \times 8 \times 8 = 2048$ opcodes, $255 \times 8 = 2040$ opcodes, and 16 opcodes, respectively, for a total of 4104 opcodes. A 12-bit word has room for only 4096, so there is no way to do it.

6. The instructions load 20, 40, 60, 30, 50, and 70, respectively.

7. The optimal code for the three machines is as follows:

```
PUSH A          LOAD E          MOV R0,E        MUL R0,E,F
PUSH B          MUL F           MUL R0,F        SUB R0,D,R0
PUSH C          STORE T         MOV R1,D        MUL R1,B,C
MUL             LOAD D          SUB R1,R0       ADD R1,A,R1
```

```
ADD          SUB T        MOV R0,B     DIV X,R0,R1
PUSH D       STORE T      MUL R0,C
PUSH E       LOAD B       ADD R0,A
PUSH F       MUL C        DIV R0,R1
MUL          ADD A        MOV X,R0
SUB          DIV T
DIV          STORE X
POP X
```

The 0-address machine has 12 opcodes and 7 addresses, for a total of 208 bits. The 1-address machine has 11 opcodes and 11 addresses, for a total of 264 bits. The 2-address machine has 9 opcodes, 7 addresses, and 11 registers for a total of 228 bits. The 3-address machine has 5 opcodes, 7 addresses, and 8 registers, for a total of 184 bits.

8. The machine needs 64 base registers, each wide enough to reference the entire memory. A 6-bit number selects a base register. By loading the registers appropriately, an arbitrary collection of 64 addresses can be accessed.

9. If the modified instruction is in a procedure that is called a second time, it will be wrong upon entry the second time. Thus an initialization mechanism is needed for each call.

10. The reverse Polish notation formulas are:

a. $A\,B + C + D + E -$
b. $A\,B - C\,D + \times E +$
c. $A\,B \times C\,D \times + E +$
d. $A\,B - C\,D\,E \times - F \,/\, G \,/ \times H \times$

11. Converting to infix, we see:

a. $(A + B) + C = A + (B + C)$
b. $(A - B) - C \neq A - (B - C)$
c. $(A \times B) + C \neq A \times (B + C)$

12. The corresponding infix formulas are:

a. $(A - B + C) \times D$
b. $(A / B) + (C / D)$
c. $A / (B \times C \times (D + E))$
d. $A + (B \times ((C + (D \times E) / F) - G) / H)$

13. Obviously, there are many answers to this question. Three examples are $A\,B + +$, $A + + B\,C$, and $A\,B\,C\,D +$.

14. The reverse Polish notation formulas are as follows:

a. A B AND C OR
b. A B OR A C OR AND
c. A B AND C D AND OR

15. The reverse Polish notation formula is $5 2 \times 7 + 4 2 / 1 + -$. The generated code is:

```
BIPUSH 5
BIPUSH 2
IMUL
BIPUSH 7
IADD
BIPUSH 4
BIPUSH 2
IDIV
BIPUSH 1
IADD
ISUB
```

16. The designers of the various assembly languages had different taste. There is absolutely no connection between the assembly language notation and the instruction set, addressing modes, or bit encoding of instructions. It would be an easy matter to write an assembler for the Pentium 4 that used the second operand as the destination, or to write an assembler for the UltraSPARC III that used the first operand as the destination. It is important to realize that the choice of the assembly language syntax is completely independent of how the bits are encoded at run time.

17. There are 5 bits per register, so there can be up to 32 registers. Theoretically, a choice for fewer than 32 is possible, but it would be a very peculiar choice.

18. The use of format 3 is inherent in the opcode. An instruction like ADD uses either format 1 or format 2, depending on bit 23, but an instruction like BEQ always uses format 3. It does not have to be told to do so.

19. There are five possible results: $X < A$, $X = A$, $A < X < B$, $X = B$, and $X > B$. To record all the possibilities, 3 bits are needed. Of the eight combinations, only five would be used.

20. It is needed when adding decimal (BCD) numbers. If we add the 16-bit hex integer 0008H to 0008H using binary add, we get 0010H. This result is incorrect for decimal numbers, since 8 plus 8 is 16, and 16 is represented in BCD as 0016H. The carry out of bit 3 allows the AAA instruction to figure out what happened and fix the result.

21. *SETHI* sets bits 10 through 31, while *ADD* sets bits 0 through 12. There is overlap here, which means that bits 10, 11, and 12 are set by both of them. Since *ADD* does sign extension, it is best to make sure bit 12 of *ADD* is 0. Still, bits 10 and 11 can be set by both instructions. Thus there are four combinations of who does it. (Of course, if both bits are 0, it is hard to say which 0 was a data bit and which was a placeholder.)

22. The idea is pointless, since all that matters is the number of bits. How they are positioned in the word or whether one chooses to regard them as one, two, or more fields makes no difference. Tell your friend to get a good night's rest and start all over the next day.

23. Memory above 255 is addressed using the 16-bit *DPTR* register, which is first loaded with the address of the byte needed.

24. Four fields are needed: two operands, a condition, and a branch address. Within this general framework, there are many possible solutions. A 3-bit number is sufficient for the condition. The branch field could be absolute (a machine address), relative (a displacement), or a general operand with mode and register.

25. The five numbers are as follows:

a. 0000 1001 0101 1100
b. 1111 1001 0101 1100
c. 0101 1100 0011 0000
d. 0101 1100 0011 1001
e. 0011 1001 0101 1100

26. It obviously depends on the machine, but typically you would generate a zero in a register and then store it in memory. You can generate a zero in a register by loading it from memory, by subtracting a number from itself, or by EXCLUSIVE ORing a number with itself.

27. The answer is 1101 0000 0010 0011.

28. The solution is clear once you realize that A XOR A = 0 for any A, so XORing anything into a variable twice does not change the variable. The following three steps performed in sequence do the job.

(1) B = A XOR B
(2) A = A XOR B
(3) B = A XOR B

29. The constant must have not more than two 1 bits in it (e.g., 11000000 or 00100010 but not 00100011).

30. For the Pentium 4 we have:

a. `MOV EAX,4; MOV [EBP+i],EAX` (5 bytes)
b. `MOV EAX,[EBP+j]; MOV [EBP+i],EAX` (6 bytes)
c. `MOV EAX,[EBP+j]; DEC EAX; MOV [EBP+i],EAX` (7 bytes)

For the UltraSPARC III we have:

a. `MOV 4,%O0; ST %O0,[%FP-i]` (8 bytes)
b. `LD [%FP-j],%O0; ST %O0,[%FP-i]` (8 bytes)
c. `LD [%FP-j],%O0; ADD %O0,–1,%O1; ST %O1,[%FP-i]` (12 bytes)

For IJVM we have:

a. `ICONST_4; ISTORE_0` (2 bytes)
b. `ILOAD_1; ISTORE_0` (2 bytes)
c. `ILOAD_1; ICONST_1; ISUB; ISTORE_0` (4 bytes)

31. The instruction might contain two register numbers, a 3-bit condition field, and a branch offset. The conditions could be less than, less than or equal, equal, etc. If the condition held between the two registers, the branch would be taken. Otherwise the instruction would do nothing. Many simple while loops could use this instruction.

32. To solve this, we need to know how many moves it takes to solve the problem for n disks. Let the number of moves needed for n disks be S_n. Then by inspection of the code we have

$$S_n = S_{n-1} + 1 + S_{n-1}$$

because to solve the problem for n, we have to solve it twice for $n - 1$ plus one extra move. Thus we now know that $S_n = 2S_{n-1} + 1$. For $n = 1$, we know that S_n is 1. Thus for succeeding values of n we have $S_n = 3, 7, 15, 31, 63, 127, 255, 511, 1023$, etc. Thus it is clear that $S_n = 2_n - 1$. For 64 disks, we need $2^{64} - 1$ moves. At 60 sec per move, this is about 1.1×10^{21} sec. A year has about 3.15×10^{7} sec, so we are looking at about 3.5×10^{13} years. This is about 3500 times the estimated age of the universe.

33. When an I/O device wants to signal an interrupt, it asserts the interrupt line. In some cases it may also assert lines giving the priority. If the device did not provide vector information, how would the CPU know which vector to use? If priority data are also present, that would help a little, but what would happen if there were more devices than priority levels? The vector information is essential and cannot be replaced by a table.

34. The disk has 32 KB per track, which is 16K words. Data transfer runs at 16K transfers per 16 msec, which requires 8.192 msec of bus cycle time per 16 msec real time. This leaves 7.808 msec per 16 msec for the CPU, thus reducing it to 7.808/16 or 48.8 percent of its speed (i.e., it is slowed down by 51.2 percent). The number of bus cycles per instruction is irrelevant.

35. With normal procedures, in the absence of interrupts, calls are made in the order dictated by the program flow. Each procedure runs until it completes, then it returns. The issue of ‘‘Should I stop running this procedure and start running some other one?’’ does not occur. With interrupts it does, so some way is needed to answer the question and priorities are one way to do this.

36. The value of many registers is not directly related to predication. It is related to the need to avoid dependences and thus avoid register renaming. If there are lots of spare registers, the compiler can avoid some dependences by just using new registers all the time.

37. They do not exist. A speculative instruction is one that is executed before it is known that the variable being loaded will really be needed. For a *LOAD* at worst, some time is wasted fetching it, but no damage is ever done. With a speculative *STORE*, a memory word is being overwritten. If the path it is on is not taken, there is no way to recover the contents of the lost memory word.

38. The bridge can handle 4000 packets per second, 2000 from each network. At 1 KB per packet, neither network can operate above 2 MB/sec.

39. It has to know how many bytes of storage are occupied by the parameters so that it can find the stored return address and old frame pointer. In a strongly typed language, like Java, this information is known at compile time.

SOLUTIONS TO CHAPTER 6 PROBLEMS

1. It would be wasteful to have the operating system interpret instructions such as ADD that the microprogram can handle perfectly well. The additional level of interpretation is omitted for efficiency reasons.

2. The address space is 2^{32} bytes. Each page is 4 KB, so the number of pages is $2^{32}/4096$ or 2^{20}, which is 1,048,576.

3. In theory, virtual memory is just a mapping between virtual and physical addresses, so any page size could be used. However, normal MMU hardware uses some of the virtual address bits as an index into the page table. With pages whose size was not a power of two, this could not be done. Thus it is theoretically possibly, just not practical.

4. a. These addresses cause faults: 2048 to 4095, 5120 to 6143, 7168 to 8191.
b. 3072, fault, 4095, 1024, 1025, fault, 2048.

5. a. With LRU, memory looks like this (* indicates a page fault):

```
, , , 0 *
, , 0, 7 *
, 0, 7, 2 *
```

```
  , 0, 2, 7
0, 2, 7, 5 *
2, 7, 5, 8 *
7, 5, 8, 9 *
5, 8, 9, 2 *
8, 9, 2, 4 *
```

b. With FIFO, memory looks like this (* indicates a page fault):

```
  ,  ,  , 0 *
  ,  , 0, 7 *
  , 0, 7, 2 *
  , 0, 7, 2
0, 7, 2, 5 *
7, 2, 5, 8 *
2, 5, 8, 9 *
2, 5, 8, 9
5, 8, 9, 4 *
```

6. Maintain a global counter that is incremented at each page fault. When a new page is brought in, associate with it the current value of the counter. The page with the lowest number is then the oldest. Note that wraparound must be avoided, so 32-bit numbers should be used.

7. If the page fault handler were not present, an attempt to access it would cause a (recursive) page fault. Handling of this fault would cause another fault, and so on. The machine would loop forever. To prevent it, the page fault handler must be marked in some way to prevent its removal.

8. When a page is loaded, it should be set to read + execute. Any attempt to write to it will cause a trap. The operating system then notes that the page is dirty, turns on write permission for the page, and restarts the previous instruction.

9. The 10 accesses give the following results: (1) 6145, (2) 10, (3) page fault, (4) protection fault, (5) 2050, (6) 28686, (7) 16484, (8) page fault, (9) segment fault, (10) protection fault. Note that if a protection fault occurs, it does not matter if the page table and page are in memory.

10. Yes. If a disk transfer is part way through, and suddenly the program is moved, the rest of the transfer is likely to go to the wrong place and wreak havoc. If the I/O device uses virtual addresses instead of real addresses, there is some hope (although it is very tricky), but if I/O uses physical addresses, it is impossible to get it right.

11. If a file could be mapped into the middle of page, a single virtual page would need two partial pages on disk to map it. The first page, in particular, would be mapped onto a scratch page and also onto a file page. Handling a page fault for it would be a complex and expensive operation, requiring copying of data. Also, there would be no way to trap references to unused parts of pages. For these reasons, it is avoided.

12. The usage pattern of the Pentium 4 is such that most references use one of the segment registers. When a segment register is loaded, this is a signal to the processor that the corresponding segment is going to be used soon, so fetching the descriptor into a register and keeping it there means that on subsequent references, the LDT or GDT does not have to be accessed.

13. The linear address is 18000. This means that page directory entry 0 is used, with page 4 and offset $18000 - 4 \times 4096 = 1616$.

14. FIFO and LRU can be used, but other possibilities also exist. For example, each segment may be bounded by two holes (except possibly the first and last segments). When a segment is removed, the two adjacent holes and the space previously occupied by the segment are merged into a single hole. One algorithm would inspect each segment and determine the size of the hole created by its removal, choosing the segment to remove on that basis, for example, the largest new hole or the largest new hole to segment-removed ratio.

15. Internal fragmentation occurs when the last page of a segment is not full. External fragmentation is the phenomenon of holes interleaved between segments. Internal fragmentation can be helped by using small pages. External fragmentation can be helped by better segment placement or compaction.

16. LRU probably would work fine. If the store has not sold some item for a couple of years, it is probably safe to drop it. FIFO, on the other hand, would probably be much worse since the oldest product in the assortment is likely to be a basic foodstuff such as flour, salt, sugar, or coffee.

17. There is no startup time to fetch a cache block so doubling the cache block size doubles the access time. It takes twice as long to fetch 128 bytes as to fetch 64 bytes. Put in other terms, there is no advantage to fetching 128 bytes over fetching 64 bytes twice. With paging, there is a long startup time before accessing a page, often 20 msec or more, depending on the disk. Here fetching 2 KB twice is much slower than fetching 4 KB once. Thus there is a big advantage to fetching large units at once to amortize the startup time. This advantage drives the page size up. No comparable effect drives cache block size up, and making cache blocks too large may be wasteful as the data fetched may never be used.

18. When an open system call is done, the operating system fetches information from the disk telling where to find the blocks of the file. This information (e.g., the UNIX i-node) is kept in main memory until the file is closed. Also, access control is done only once, not on each reference to the file.

19. The disk has $800 \times 5 \times 32 = 128,000$ sectors. With a bit map, 128,000 bits or 4000 words are needed, independent of the current number of free sectors. With fewer than 4000 holes, the free list wins, but with more than 4000 holes, the bit map wins.

20. The mean size of a data block or a hole can be found by computing the sum $\sum i\, 2^{-i}$, where i runs from 1 to ∞. The sum can be done starting from the well-known formula for the sum of a geometric series:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

If we differentiate each side of the equation with respect to x, we get

$$\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$$

If we now multiply both sides by x, we get a sum that is almost what we need, except that the lower limit is 0 instead of 1. However, the zeroth term makes no contribution to the sum, so the lower limit can be replaced by 1 without changing the result. For $x = 1/2$, we find that the average data block or hole is two sectors, so the average data block + hole pair is four sectors. The number of such pairs, and hence the number of holes, is then $N/4$.

21. It is very unlikely. The problem is: How far apart should the operating system place the files as they are created? It has no way of knowing.

22. Since the total volume occupied by all the small files is a tiny fraction of the disk, doubling it or more will have little effect on the amount of disk space used. But large disk blocks will greatly increase performance, especially on large files. Thus this data suggests using a large disk block, say 8 KB.

23. If there is one CPU, disabling interrupts works. If there are two CPUs, it fails, because a CPU can disable only its own interrupts, not those of the other CPU.

24. With three or more processes it clearly fails. If P1 inspects the semaphore and finds it 1, before it can do a down either P2 or P3 could sneak in and do a down first. With only two processes it is a little more complicated. In principle, the same race exists. P1 could inspect the semaphore and find it 1, but before doing a down P2 might do a down. However, this situation occurs only when both P1 and P2 want to do a down simultaneously. If that situation occurs, both processes will sleep forever (a deadlock). In other words, the race occurs only when the programs are logically incorrect. With three processes there is no danger if P1 and P2 both want to sleep at the same time because P3 can always wake them up later.

25. The behavior of the semaphore (S) and the processes (P1, P2, and P3) are given in the following table.

Time	*S*	*P1*	*P2*	*P3*
0 – 100	1	R	R	R
100 – 200	0	R	R	R
200 – 300	0	S	R	R
300 – 400	0	R	R	R
400 – 500	0	R	R	S
500 – 600	0	S	R	S
600 – 700	0	R	R	S
700 – 800	0	R	S	S
800 – 900	0	R	R	S
900 –1000	0	R	R	R

26. One way to solve the problem is to associate with each file a semaphore, BUSY, initially 1. When a process wants to use a file, it does a down on the semaphore. The first process doing the down will be granted accesses, but subsequent processes will block. When the successful process is done, it does an up, making the file available again.

27. The procedure *lock* is

```
LOCK:   TSL X
        JMP LOCK
        RET
```

In other words, just keep checking until it is unlocked then return. *unlock*(x) merely stores a zero in X.

28. The *in* and *out* pointers after each operation are as follows:

	in	*out*
a.	22	0
b.	22	9
c.	62	9
d.	62	26
e.	9	26
f.	9	6
g.	17	6
h.	17	17

29. The first 10 disk addresses are in the i-node. The single indirect block holds 512 more. The double and triple indirect blocks hold 262,144 and 134,217,728 blocks, respectively. The total number of blocks is thus 134,480,394. With 2 KB data blocks, this gives a maximum file size of 274,877,906,944 bytes.

30. The only change is that the entry *game3* in the directory */usr/ast/bin* is deleted. The file is still accessible via */usr/jim/jotto*, however.

31. I/O redirection does not involve pipes, just the fact that open files are inherited across fork and exec, so there is no problem here. Pipelines do use pipe, but the shell could easily be modified to run the filters sequentially instead of in parallel, with the intermediate files being buffered on the disk, assuming that sufficient disk space is available.

32. This proposal will make redirection cease to work. At present, the child process forked off by the shell closes 0 or 1 if redirection is required and then opens the redirected file. If the new file descriptor were different from the old one, the redirection would not be transparent and would fail.

33. The access control list entries are processed in order. As soon as a match is found, the search stops. In this example, the ACL would say: Roberta: no access; *: full access. As soon as the system found that the first entry matched, it would stop searching and never even see the second entry granting everyone full access.

34. (1) Threads within a process; (2) two processes sharing a mapped file.

SOLUTIONS TO CHAPTER 7 PROBLEMS

1. The three cases are as follows:

a. Programming = 100 months, execution time = T.
b. Programming = 1000 months, execution time = 0.25T.
c. Programming = 100 + 20 = 120 months, execution time = 5T/8.

2. If the compilers produce object modules, the compilers have the same problems as the assemblers: namely, if the first statement of a procedure is a GOTO, the compiler does not yet know the absolute address of the label. On the other hand, if assembly code is produced, the compiler does not have to deal with the problem. It is pushed off onto the assembler.

3. A way would have to be found to pacify a lot of angry users who were used to the other convention. However, technically, the order is completely arbitrary. The machine architecture itself does not dictate, or even vaguely suggest, one assembly language syntax over the other. It is the same as dividing the hour into 60 parts instead of 100 (metric hour). Somebody made this choice 6000 years ago, and now we are all stuck with it.

4. No. On pass one S is defined; on pass two R is defined. It would take two more passes to define P.

5. Symbols can be up to 10 characters, encoded in radix 25. Let the 10-character symbol be $C_9\, C_8\, C_7\, C_6\, C_5\, C_4\, C_3\, C_2\, C_1\, C_0$. The value of the word representing the symbol is $\sum 25^i C_i$, where i runs from 0 to 9 and the Cs are encoded with A = 0, B = 1, ..., Z = 24. The largest symbol that can be produced this way is less than $2^{48} - 1$.

6. An instruction is something translated to a binary number and put into the object program for execution by the machine. A pseudoinstruction is a command to the assembler and does not generate an executable instruction.

7. The program counter is used at *run* time to keep track of which instruction to execute next. The instruction location counter is used at *assembly* time to keep track of the memory location to put the next instruction in.

8. The values of the symbols are: EVEREST = 1000, K2 = 1001, WHITNEY = 1002, MCKINLEY = 1004, FUJI = 1007, and KIBO = 1008.

9. It depends on how the assembler is implemented. If the symbol table and the opcode table are different, there is no problem with names being reused. However, if a single table is used for all symbols, duplicates may have to be forbidden. An assembler could permit a label to duplicate an opcode since it is normally clear from the context which is which, but doing so requires extra work.

10. If we number the cities 1 through 12, the first probe is number 6, New Haven. Berkeley < New Haven, so the new list to be searched is 1 through 5. The next probe is number 3, Cambridge. Berkeley < Cambridge, so the new list is 1 through 2. The third probe is number 1, Ann Arbor. Since Berkeley > Ann Arbor, we now are left with a list starting and ending at entry 2, Berkeley. On probe 4, we find it.

11. Sure, no problem.

12. The computation is as follows:

els = (5 + 12 + 19) mod 19 = 17
jan = (10 + 1 + 14) mod 19 = 6
jelle = (10 + 5 + 12 + 12 + 5) mod 19 = 6
maaike = (13 + 1 + 1 + 9 + 11 + 5) mod 19 = 2

jan and jelle hash to the same value, 6, by accident. Collisions like this can be accommodated by chaining all entries with the same hash code together on a linked list.

13. The probability of success on any attempt is $1 - r$. The probability, $P(k)$, of requiring exactly k attempts is the probability of $k - 1$ failures, r^{k-1}, times the probability of success, $1 - r$. The mean number of probes is found by summing $kP(k)$ from 1 to n. The result is

$$\text{mean number of probes} = \frac{1 - r^{n+1}}{1 - r}$$

14. Each CPU should hold one symbol and the corresponding value in its local memory. When the main processor wants to look up a symbol, it puts the symbol in the first of the shared registers and sets a second register to 1, indicating that a search is desired. All the CPUs then check to see if they have the needed symbol. If any one of them finds the symbol, it puts the value in the third register and clears the 1 in the second register. After one lookup time has passed, the main processor inspects the three registers to see what happened.

15. Multiple ILCs are needed, one for each segment. Each one starts at 0. They are maintained and incremented independent of one another.

16. If this is the only program using the DLLs and it always needs all of them, yes, it would be more efficient to combine them. However, if many programs use the DLLs, as is often the case, and different programs need different combinations, breaking them up may make sense so that not all the procedures need be in memory at the same time if they are not all needed.

17. If a DLL starts out with `BR 100` instruction and this DLL is mapped to address 1000 in one process and address 2000 in another process, it will fail in both of them. One way out is to make sure all the code in all DLLs is position independent. Another way is to give each DLL a unique virtual address and have it always map there. Finally, on a segmented architecture, each DLL can be put in a separate segment.

18. It could fail if a procedure pulled out of the library itself needs another procedure in the library and the needed procedure has already been passed. One solution is to make multiple passes over the library. Another is to require it to be sorted so that there are no backward references. Finally, a third solution is to put an index at the front of the library telling what every procedure needs, so the complete list is known before the scan starts.

19. Sure. The macro assembler is just doing string substitution. It does not care at all whether the parameters are variables, registers, or anything else. During macro expansion it is just moving characters around. If the resulting assembly code is legal, everything will work fine.

20. It means you will need three passes: one to collect the macro definitions, one to expand the macros and build the symbol table, and one to generate code.

21. Define a macro, LOOP, and inside the macro body call LOOP. Each expansion leads to another one, in a never-ending cycle. Assembly will terminate only when the assembly runs out of memory for keeping track of where it is.

22. The constants are 0, 200, 1000, 1600, and 2100, respectively.

SOLUTIONS TO CHAPTER 8 PROBLEMS

1. No. The essence of a VLIW instruction is that there are multiple operations that can be started independently. The Pentium does not have this.

2. The clipped values are 96, 0, 255, and 255.

3. (a) No, because loads are prohibited in slot 3.
(b) Yes.
(c) No, because floating adds may not be done in slot 2.

4. In the first one (fine-grained multithreading), the rotation continues, so we get *A4*, *B4*, and *C4*. In the second one (coarse-grained multithreading), we get *B2*m *C5*, and *C6*.

5. If it takes *k* cycles in all to handle a level 1 cache miss, we need (at least) *k* threads so that it takes *k* cycles to run through them all. After *k* cycles have elapsed, the thread that missed will run again, and by now the memory word needed will be present.

6. It is MIMD. Although the general work order is the same, each bee looks for its own flowers (has its own instruction stream). There is no lockstep coordination at all.

7. If a program expects a sequentially consistent memory and cannot live with anything less, the memory must provide sequential consistency. However, to improve performance, some memory systems provide a weaker model. It is then essential that the software agrees to abide by the rules imposed by this model. Generally, it means that programs obeying the rules will perceive what looks like a sequentially consistent memory.

8. No big deal. We have seen many bus arbitration schemes in Chap. 3. The bus arbiter decides who goes first. Then we have sequential access.

9. *Write through* would not be affected since it only snoops on the address lines.

10. In 64 instruction times, each CPU gets the bus once, allowing it to carry out about four instructions, so the total number of instructions executed in 64 instruction times is 256. A single CPU could get the bus 16 times in this period and thus carry out 64 instructions. The 64-CPU system is only four times faster than the 1-CPU system.

11. The I state is essential since the cache is empty when the CPU is booted. The M state is needed if dirty data are to be allowed in the cache, otherwise the protocol would no longer be a write-back protocol. The S state could be sacrificed, in which case a block could only be in one cache at a time, that is, always in exclusive state. Dropping S would mean that when a second CPU read a block, the first one would have to abandon it. This is doable but hurts performance for read-only data. Alternatively, the E state could be sacrificed. This would mean that even clean data located in only a single cache would be in S state. A write to these data would then require a bus transaction telling other caches to invalidate the block. A write to an E block does not need a bus transaction. All in all MSI is probably a better deal than MEI because sharing occurs a lot, and the cost of an extra bus transaction once in a while is bearable.

12. For read hits, no bus transactions are ever needed, but for a write hit on a block in S state, a bus transaction is needed to invalidate other caches in order to convert the local block from S state to M state.

13. a. The probability that any machine is silent is $1 - p$, so the probability that all n are silent is just $(1 - p)^n$.

b. The probability that machine 0 is the only requestor is $p(1 - p)^{n-1}$. Since all n machines are equally likely to transmit, the probability that any one of them does it is $np(1 - p)^{n-1}$.

c. The probability of two or more is just 1 minus the above two expressions, or $1 - p(1 - p)^{n-1} - (1 - p)^n$.

14. There are five per boardset, so with 18 boardsets, the total is 90.

15. CPUs 000, 010, 100, and 110 are cut off from memories 010 and 011.

16. It depends on how fast the memory is. If the memory is fast, and all requests are handled without the requesting CPU releasing the bus, then it does not matter what address is accessed and there are no hot spots. On the other hand, if memory is slow and there are many memory modules that can act in parallel (i.e., a CPU releases the bus after making a request), then having many requests to the same module is a problem

17. There are 12 stages to be crossed in each direction, for a total delay of 120 nsec. Thus two delay slots are needed.

18. Just connect processor i and memory i with a wire. In that way, these requests do not have to use the switching network. Topologically, one can think of the whole omega network as being wrapped around the outside of a cylinder, so that processor i and memory i are now adjacent. A processor can issue a request to the right, that is, using the switching network, or to the left, directly to its own memory. The disadvantage is that the memories must now be able to handle two simultaneous requests, one from each side. As a result, the memory circuitry becomes more complicated and expensive.

19. If the page remains remote during execution the $N/100$ references will take $1.2N$ nsec. If it is made local, the access time will be $C + 0.2N$ nsec, including the time to copy it. Thus if $C + 0.2N < 1.2N$, the page should be copied; otherwise, the time to copy it is too much to make the copy worthwhile. For $C < N$ it is worth copying.

20. Each node has 2^{17} directory entries. Since there are 512 nodes, a directory entry is now 10 bits, 1 bit to mark a block as cached or not and 9 bits to tell where it is. Thus the total size of the directory at each node is 1,310,720 bits. The memory at that node is 8 MB or 67,108,864 bits. The percent overhead is therefore about 1.95 percent. This overhead is slightly larger than in the figure due to the larger field needed to specify a node.

21. (a) 2. (b) 1. (c) 6. (d) 4. (e) 6. (f) 4. (g) 3 (h) 4.

22. (a) 0. (b) 6. (c) 0. (d) 1. (e) 1. (f) 3. (g) 2. (h) 3.

23. On a grid, the worse case is nodes at opposite corners trying to communicate. However, with a torus, opposite corners are only two hops apart. The worst case is one corner trying to talk to a node in the middle. For odd k, it takes $(k - 1)/2$ hops to go from a corner to the middle horizontally and another $(k - 1)/2$ hops to go to the middle vertically, for a total of $k - 1$. For even k, the middle is a square of four dots in the middle, so the worst case is from a corner to the most distant dot in that four-dot square. It takes $k/2$ hops to get there horizontally and also $k/2$ vertically, so the diameter is k.

24. The network can be sliced in two by a plane through the middle, giving two systems, each with a geometry of $8 \times 8 \times 4$. There are 64 links running between the two halves, for bisection bandwidth of 64 GB/sec.

25. Amdahl's law states: speedup = $n/[1+(n - 1)f\,]$. As $n \to \infty$, the 1s in the denominator become negligible, and the speedup becomes n/nf which is $1/f$. If $f = 0.1$, no matter how many CPUs you use, you will not be able to make the program run more than 10 times faster. For example, it the program initializes itself for 6 sec, then works for 54 sec, even if you get the work time down to 0 sec, you will not be able to get the running time below 6 sec, that is, a 10-fold speedup.

26. The average bandwidth per CPU for the two bus-based systems is $b/4$ bps and $b/16$ bps, respectively. For a 64-CPU system, it drops to $b/64$ bps. For an infinite number of CPUs, it drops to 0. For the two grid-based systems shown in the figure, the average bandwidth per CPU is b bps and $1.5b$ bps, respectively. For a 64-CPU system in an 8×8 grid, there are $2 \times 7 \times 8$ links, giving $112b/64$ bps. As the number of CPUs goes to infinity, the bandwidth per CPU tends to $2b$ because adding each new CPU also adds two new links.

27. One variant is to have the system copy the message buffer into the kernel, then release the sender. When the message is actually sent then becomes irrelevant. This scheme allows the sender to continue earlier than the actual transmission, which may be important if there is a long queue waiting for the network. On the other hand, it involves extra copying (to the kernel), and each extra copy costs performance. In the original variant, it may well be possible to do the actual transmission from the user buffer, eliminating the copy.

28. Reads on replicated objects are done locally, with no network traffic, so having a large number of them is very efficient. Writes require network traffic, so they are slow when objects are replicated.

SOLUTIONS TO APPENDIX A PROBLEMS

1. The numbers are: 11111000000, 111110100000, and 10000000000000, respectively.

2. In decimal it is 617, in octal it is 1151, and in hex it is 269.

3. All strings containing only the 10 digits and the letters A, B, C, D, E, and F are valid hex numbers. All the examples given are valid except BAG.

4. The answers are: 1100100, 10201, 1210, 400, 244, 202, 144, and 121.

5. The number of valid strings of length k with r possibilities in each position is r^k.

6. With just your hands you have 10 bits, so you can count to 1023. With hands and feet the limit becomes $2^{20} - 1 = 1{,}048{,}575$. In two's complement the range is $-524{,}288$ to $+524{,}287$.

7. The results are: 10011100, 11111110, 00000001, and 00000000. The first calculation results in an overflow. The subtractions are done by forming the two's complement of the subtrahend and then adding it to the minuend.

8. The results are: 10011100, 11111111, 000000000, and 11111111. Again the first calculation gives an overflow. The subtraction is done by forming the one's complement of the subtrahend and adding it to the minuend.

9. The five sums are: 001, 111, 101, 011, and 000, respectively. The NZV condition code bits are 000, 100, 100, 000, and 011, respectively.

10. The numbers are: 006, 997, 100, 985, 998, and 000.

11. The rule for addition of nine's complement numbers is analogous to that for the addition of one's complement numbers: the carry is added to the number. The sums are: 0001, 9999, 9994, and 0044.

12. It is similar to two's complement: just do the addition and ignore the carry.

13. The tables are: $0 \times 0 = 0$, $0 \times 1 = 0$, $0 \times 2 = 0$, $1 \times 0 = 0$, $1 \times 1 = 1$, $1 \times 2 = 2$, $1 \times 3 = 3$, $2 \times 1 = 1$, and $2 \times 2 = 11$

14. The answer is 10101 ($7 \times 3 = 21$).

SOLUTIONS TO APPENDIX B PROBLEMS

1. The number 9 can be written as 1.001×2^3. Adding the bias of 127 to the exponent, we get

0 10000010 00100000000000000000000

In hexadecimal, this is 41100000H. Similarly, 5/32 is 3E200000H, −5/32 is BE200000H, and 6.125 is 40C40000H.

2. Writing 42E48000 in binary we get

0 10000101 11001001000000000000000

Subtracting off the bias of 127 from the exponent, we see that the unbiased exponent is 6, so we have 1.11001001×2^6, which is 114.25. Similarly, parts (b) through (d) yield 1.0625, 2^{-126}, and −122,880, respectively.

3. To convert to the floating-point format, first convert 7/64 to binary: 0.000111. One representation is

0 1000000 000111000000000000000000

Since the radix of exponentiation is 16, shifts of the fraction must be done 4 bits at a time. No such shifts are possible, so the above number is already normalized. In hex it is 401C0000H.

4. To normalize, shift left 1 bit at a time, adding 1 to the exponent at each step, until the leftmost bit of the fraction is 1. The results are

(a) 0 1000011 1010100000001000
(b) 0 1000101 1111111111000000
(c) 0 1000011 1000000000000000

The third one is already normalized.

5. Writing the two operands out in binary we get

0 01111101 11000000000000000000000 (= 7/16)
0 01111011 00000000000000000000000 (= 1/16)

Making the implicit 1 and the binary point explicit, we get

1.11×2^{-2}

1.00×2^{-4}

Adjusting exponents, we get

1.11×2^{-2}

0.01×2^{-2}

Since the exponents are now the same, we can add the two fractions. The result has the same exponent as the operands. The result is: 10.00×2^{-2}. When normalized to IEEE format, this becomes 1.00×2^{-1} (1/2), which in hex is 3F000000H.

6. The smallest and largest numbers are

Model 0.001: 0 0000000 10000000 0 1111111 11111111
Model 0.002: 0 00000 1000000000 0 11111 1111111111

In decimal, the Model 0.001's limits are about 2.7×10^{-20} and 9.2×10^{18}. The Model 0.002's limits are about 7.6×10^{-6} to 3.3×10^{4}. The number of decimal digits in an n-bit fraction can by found be solving the equation $2^n = 10^x$ for x. For Model 0.001, $n = 8$, and approximately $x = 2.4$. For Model 0.002, $n = 10$, and approximately $x = 3.0$. As far as buying either one goes, it is hard to imagine an application that can live with 32768 as the largest floating-point number, so Model 0.002 is awfully hard to swallow. But with only 2.4 digits, Model 0.001 is not good for much either. I would buy a machine with at least 32 bits for floating-point numbers.

7. Subtraction of two nearly equal numbers. Consider decimal numbers with 5 digits worth of significance. If we subtract 0.21345 from 0.21347 we get 0.00002. Even representing this as 0.2×10^{4}, we still only have one significant digit of result.

8. An approximation of the square root can be made by dividing the exponent in half, for example, by shifting it 1 bit to the right. For example, in decimal an approximate square root of 4×10^{8} is 4×10^{4}. The same holds for binary numbers.

SOLUTIONS TO APPENDIX C PROBLEMS

1. The values of *AH* and *AL* are 2 and 190, respectively.

2. The 20-bit address implied by *CS* is 64. Thus, the code segment runs from 64 through and including 65,599.

3. The largest value possible in a segment register is 65,535, which corresponds to a segment base address of 1,048,560. A maximum-sized segment starting there would end at 1,114,095. Note that this address is actually more than 2^{20}.

4. a. The code segment begins at address 640, so the next instruction is at 660.
b. The word referenced is at $8000 \times 16 + 2$, or 128,002.

5. The return address is at `BP + 2` and the first argument is as `BP + 4` so the instruction is `MOV AX,4(BP)`.

6. No. The difference between two labels in the same section is a constant, independent of where they occur in the section. Adding lines before them does not make any difference. The difference between two labels remains unchanged. Adding two labels is a totally different story. If, say, *n* bytes worth of data are included before the first label, the sum $de + hw$ will increase by $2n$. It is hard to imagine a program in which this behavior would be acceptable.

7. The code is

```
MOV AX,a
ADD AX,b
ADD AX,2
MOV x,AX
```

8. The code is

```
PUSH y
PUSH x
CALL foobar
```