

6

Solutions

6.1 If the time for an ALU operation is 4 ns, the cycle time would be 10 ns in the single-cycle implementation. The cycle time for the pipelined implementation would be 4 ns and the speedup obtained from pipelining the single-cycle implementation would only be $10/4 = 2.5$.

6.2 The diagram should show three instructions in the pipeline. A forwarding path is needed from the second instruction to the third instruction because of the dependency involving \$4.

6.3 Obviously either the load or the `addi` must occupy the branch delay slot. We can't just put the `addi` into the slot because the branch instruction needs to compare \$3 with register \$4 and the `addi` instruction changes \$3. In order to move the load into the branch delay slot, we must realize that \$3 will have changed. If you like, you can think of this as a two-step transformation. First, we rewrite the code as follows:

```
Loop: addi $3, $3, 4
      lw   $2, 96($3)
      beq  $3, $4, Loop
```

Then we can move the load into the branch delay slot:

```
Loop: addi $3, $3, 4
      beq  $3, $4, Loop
      lw   $2, 96($3)           # branch delay slot
```

6.4 The second instruction is dependent upon the first (\$2). The third instruction is dependent upon the first (\$2). The fourth instruction is dependent upon the first (\$2) and second (\$4). All of these dependencies will be resolved via forwarding.

6.5 Consider each register:

- IF/ID holds PC+4 (32 bits) and the instruction (32 bits) for a total of 64 bits.
- ID/EX holds PC+4 (32 bits), Read data 1 and 2 (32 bits each), the sign-extended data (32 bits), and two possible destinations (5 bits each) for a total of 138 bits.
- EX/MEM holds PC target if branch (32 bits), Zero (1 bit), ALU Result (32 bits), Read data 2 (32 bits), and a destination register (5 bits) for a total of 102 bits.
- MEM/WB holds the data coming out of memory (32 bits), ALU Result (32 bits), and the destination register (5 bits) for a total of 69 bits.

6.6 No solution provided.

6.7 No solution provided.

6.8 No solution provided.

WB: The control signals in the WB stage show WB stage control bits of 11. This occurs for a `lw` instruction only. The destination of the register is \$15 shown on the `Write_register` signal line. The register address, the immediate value used to compute the effective address in the ALU, and the effective address itself were not sent further in the pipeline than the stages ID, EX, and MEM, respectively, so they are unavailable. Thus, the instruction in WB can be determined to no greater extent than

```
lw    $15, ? (?)
```

6.10 No solution provided.

6.11 In the fifth cycle of execution, register \$1 will be written and registers \$11 and \$12 will be read.

6.12 The forwarding unit is seeing if it needs to forward. It is looking at the instructions in the fourth and fifth stages and checking to see whether they intend to write to the register file and whether the register written is being used as an ALU input. Thus, it is comparing $8 = 4? 8 = 1? 9 = 4? 9 = 1?$

6.13 The hazard detection unit is checking to see whether the instruction in the ALU stage is a `lw` instruction and whether the instruction in the ID stage is reading the register that the `lw` will be writing. If it is, it needs to stall. If there is a `lw` instruction, it checks to see whether the destination is register 11 or 12 (the registers being read).

6.14 $5 + 99 \times 2 = 203$ cycles to execute the instructions, $CPI = 2.03$.

6.15 It will take 8 cycles to execute the code, one of which is a bubble needed because of the dependency involving the subtract instruction and the load.

6.16

Input name	# of bits	Comment
ID/EX.RegisterRs	5	Names an operand, current value may exist in pipeline, superseding value in Register file
ID/EX.RegisterRt	5	Names other operand
EX/MEM.RegisterRd	5	Names the destination of a result in the pipeline
EX/MEM.RegWrite	1	Tells if the value on the Rd datapath lines is to be written to the register file and, thus, if that value should be forwarded
MEM/WB.RegisterRd	5	Names the destination of a result in the pipeline
MEM/WB.RegWrite	1	Tells if the value on the Rd datapath lines is to be written to the register file and, thus, if that value should be forwarded

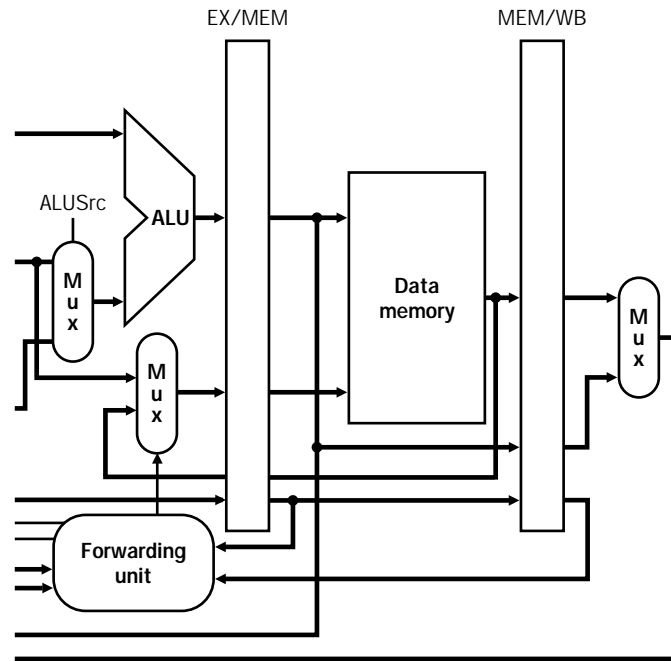
Output name	# of bits	Comment
ForwardA	2	See Figure 6.39
ForwardB	2	See Figure 6.39

6.17 No solution provided.

6.18 No solution provided.

6.19 The situation is similar in that a read is occurring after a write. The situation is dissimilar in that the read occurs much later (in the fourth cycle for load instructions vs. in the second cycle for add instructions) and that the write occurs earlier (in the fourth cycle instead of the fifth cycle). For these reasons, there is no problem. Another way of looking at it is that both the read and write occur in the same cycle (for memory access) and thus we can't possibly have a hazard!

6.20 This solution checks for the `lw-sw` combination when the `lw` is in the MEM stage and the `sw` is in the EX stage.

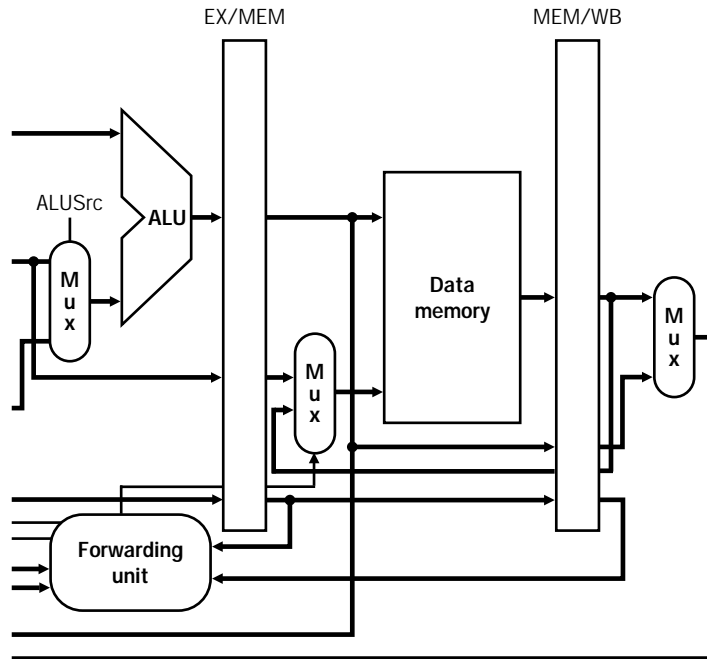


```

if (ID/EX.MemWrite and           // sw in EX stage?
    EX/MEM.MemRead and          // lw in MEM stage?
    (ID/EX.RegisterRt = EX/MEM.RegisterRd) and
                                   // same register?
    (EX/MEM.RegisterRd ≠ 0))     // but not r0?
then
    Mux = 1                       // forward lw value
else
    Mux = 0                       // do not forward

```

The following solution checks for the `lw-sw` combination when the `lw` is in the WB stage and the `sw` is in the MEM stage.



```

if (EX/MEM.MemWrite and // sw in MEM stage?
    (MEM/WB.MemToReg = 1) and MEM/WB.RegWrite and
    (EX/MEM.RegisterRd = MEM/WB.RegisterRd) and // lw in WB stage?
    (MEM/WB.RegisterRd ≠ 0)) // same register?
    // but not r0?
then
    Mux = 1 // forward lw value
else
    Mux = 0 // do not forward

```

For this solution to work, we have to make a slight hardware modification: We must be able to check whether or not the `sw` source register (`rt`) is the same as the `lw` destination register (as in the previous solution). However, the `sw` source register is not necessarily available in the MEM stage. This is easily remedied: As it is now, the `RegDst` setting for `sw` is `X`, or “don’t care” (refer to Figure 6.28 on page 469). Remember that `RegDst` chooses whether `rt` or `rd` is the destination register of an instruction. Since this value is never used by a `sw`, we can do whatever we like with it—so let’s always choose `rt`. This guarantees that the source register of a `sw` is available for the above equation in the MEM stage (`rt` will be in `EX/MEM.WriteRegister`). (See Figure 6.30 on page 470.)

A `lw-sw` stall can be avoided if the `sw` offset register (`rs`) is not the `lw` destination register or if the `lw` destination register is `r0`.

```

if ID/EX.MemRead and // lw in EX stage?
  ((ID/EX.RegisterRt = IF/ID.RegisterRs) or // same register?
  (ID/EX.RegisterRt = IF/ID.RegisterRt)) and // but not...
  not (IF/ID.MemWrite and // sw in ID stage?
  (ID/EX.RegisterRt = IF/ID.RegisterRs)) and // same register?
  (ID/EX.RegisterRt ≠ 0) // register r0?
then
  Stall the pipeline

```

Note that `IF/ID.MemWrite` is a new signal signifying a store instruction. This must be decoded from the opcode. Checking that the `lw` destination is not `r0` is not done in the stall formula on page 490. That is fine. The compiler can be designed to never emit code to load register `r0`, or an unnecessary stall can be accepted, or the check may be added, as is done here.

6.21 The memory address can now come straight from Read data 1 rather than from the ALU output; this means that the memory access can be done in the EX stage, since it doesn't need to wait for the ALU. Thus, the MEM stage can be eliminated completely. (The branch decision can be taken care of in another stage.) The instruction count will go up because some explicit addition instructions will be needed to replace the additions that used to be implicit in the loads and stores with non-zero displacements. The CPI will go down, because stalls will no longer be necessary when a load is followed immediately by an instruction that uses the loaded value. Forwarding will now resolve this hazard just like for ALU results used in the next instruction.

6.22 An `addm` instruction needs 6 steps.

1. Fetch instruction.
2. Decode instruction and read registers.
3. Calculate memory address of memory operand.
4. Read memory.
5. Perform the addition.
6. Write the register.

This means we will need an extra pipeline stage (and all other instructions will go through an additional stage). This can increase the *latency* of a single instruction, but not the *CPU throughput* because instructions will still be coming out of the pipeline at a rate of 1 per cycle (if no stalls). In fact, a longer pipeline will have a greater potential for hazards and longer penalties for those hazards, which will decrease the *throughput*. Moreover, the additional hardware for new stages, hazard detection, and forwarding logic will increase the cost.

6.23

```

lw $3, 0($5)
add $7, $7, $3 # requires stall on $3
sw $6, 0($5)
lw $4, 4($5)
add $8, $8, $4 # requires stall on $4
add $10, $7, $8
beq $10, $11, Loop

```

6.24

```

36 beqd $1, $3, 8
40 sub $10, $4, $8 # delay slot, always executed

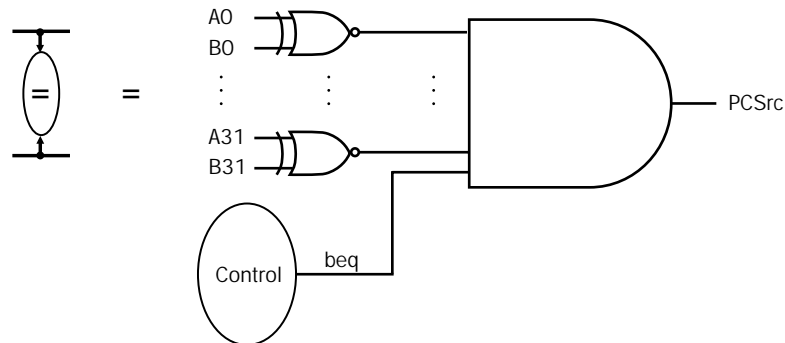
```

... the rest is the same.

6.25 No solution provided.

6.26 Flushing takes place after a taken branch. Stalling takes place if a dependency cannot be resolved by forwarding (if a `lw` is in the execute stage, a preceding dependent instruction must be stalled one cycle). If we assume that branch resolution takes place in the MEM stage, then for the sample code, in the fourth cycle of execution we have exactly this situation. The `beq` instruction will require a flush of the three `lw`, `add`, and `sw` instructions, and the `lw` instruction will require a stall of the `add` and `sw` instructions. Cooperation will take place regarding zeroing control signals. However, a conflict arises as to whether the PC should be written (flush says yes, stall says no). The flush should have priority. (A proposed solution to this problem is to change the hazard detection unit so that when it looks at the `RegWrite` signal in the EX stage, it sees the signal after it goes through the MUX used to flush the pipeline.)

6.27 See Figure 6.51 on page 499. Control in ID must be augmented to produce a signal indicating that the instruction in ID is `beq`. Register value equality is detected using a bank of XNOR gates. `PCSrc` (see Figure 6.30) is then generated in ID, making the branch decision in ID and reducing branch delay to a single instruction.



6.28

```

slt $1, $8, $9
movz $10, $8, $1
movn $10, $9, $1

```

6.29 The reason why sequences of this sort can be higher performance than versions that use conditional branching, even if the instruction count isn't reduced, is that there is no control hazard introduced, unlike with branching. Branching generally introduces stalls, which waste cycles, because the normal sequential flow of execution is disrupted. The new instructions, in contrast, are part of the normal sequential execution stream. The only hazards they introduce are normal data hazards, as with any ALU instruction, and those can be completely conquered using forwarding.

6.30 The code has been unrolled once and registers have been renamed. The question is simply how to reschedule the code for optimum performance. There are many possible solutions, one of which is

```

Loop: lw   $t0, 0($s1)
      lw   $t1, -4($s1)
      addu $t0, $t0, $s2
      addu $t1, $t1, $s2
      sw   $t0, 0($s1)
      sw   $t1, -4($s1)
      addi $s1, $s1, -8
      bne $s1, $zero, Loop

```

Regarding performance, assume `$s1` is initially `8X`. The code will require `X` iterations and a total of `11X` cycles to execute assuming branch resolution is complete in the MEM stage. If branch resolution is done in the ID stage, as shown in Figure 6.51, then the total cycles to execute would be `9X`. If the code is not unrolled (as on page 513) it will require `2X` iterations and finish on cycle $(2X) \times 9$ (assuming a one-cycle stall after the load). So, in this case, unrolling yields an improvement of $18/11 = 1.64$ times as fast.

6.31 First, we can unroll the loop twice and reschedule, assuming that `$s1` is a multiple of 12:

```

Loop: lw   $t0, 0($s1)
      lw   $t1, -4($s1)
      lw   $t2, -8($s1)
      addu $t0, $t0, $s2
      addu $t1, $t1, $s2
      addu $t2, $t2, $s2
      sw   $t0, 0($s1)
      sw   $t1, -4($s1)
      sw   $t2, -8($s1)
      addi $s1, $s1, -12
      bne $s1, $zero, Loop

```


There are many ways to modify this code so that it still works correctly if `$s2` is not a multiple of 12. Probably the best method would involve determining `$s2 mod 12` before executing the loop. One of the more simple solutions appears below. In this case we simply handle the extra cases at the end, and detect them by subtracting 12 from `$s1` before we start:

```
Loop:   addi $s1, $s1, -12
        bltz $s1, Finish
        lw  $t0, 12($s1)
        lw  $t1, 8($s1)
        lw  $t2, 4($s1)
        addu $t0, $t0, $s2
        addu $t1, $t1, $s2
        addu $t2, $t2, $s2
        sw  $t0, 12($s1)
        sw  $t1, 8($s1)
        sw  $t2, 4($s1)
        bne $s1, $zero, Loop
        j  Done
Finish: lw  $t0, 12($s1)    # $s1 may be -4 or -8
        addu $t0, $t0, $s2 # This handles 8 (if -4) or 4 (if -8)
        sw  $t0, 12($s1)
        addi $s1, $s1, 4   # $s1 is now 0 or -4
        bne $s1, $zero, Done # if $s1 is not 0 came in at -8
        lw  $t0, 4($s1)
        addu $t0, $t0, $s2
        sw  $t0, 4($s1)
Done:   ...
```

6.32 No solution provided.

6.33 No solution provided.

6.34 No solution provided.

6.35 No solution provided.

7

Solutions

7.1–7.6 The key features of solutions to these problems:

- Low temporal locality for data means accessing variables only once.
- High temporal locality for data means accessing variables over and over again.
- Low spatial locality for data means no marching through arrays; data is scattered.
- High spatial locality for data implies marching through arrays.
- Low temporal locality for code means no loops and no reuse of instructions.
- High temporal locality for code means tight loops with lots of reuse.
- Low spatial locality for code means lots of jumps to far away places.
- High spatial locality for code means no branches/jumps at all.

7.7

Reference	Hit or miss
1	Miss
4	Miss
8	Miss
5	Miss
20	Miss
17	Miss
19	Miss
56	Miss
9	Miss
11	Miss
4	Miss
43	Miss
5	Hit
6	Miss
9	Hit
17	Hit

Here is the final state of the cache:

Block #	Address
0	
1	17
2	
3	19
4	4
5	5
6	6
7	
8	56
9	9
10	
11	43
12	
13	
14	
15	

7.8

Reference	Hit or miss
1	Miss
4	Miss
8	Miss
5	Hit
20	Miss
17	Miss
19	Hit
56	Miss
9	Miss
11	Hit
4	Miss
43	Miss
5	Hit
6	Hit
9	Miss
17	Hit

Here is the final state of the cache:

Block #	Starting address
0	16
1	4
2	8
3	

7.9 There are 4K entries and each entry needs 128 (data) + 16 (tag) + 1 (valid) = 145 bits. This results in 593,920 bits, or 74,240 bytes. A bit more than the 64-KB cache size as described in the figure caption!

7.10 Simply extend the comparison to include the valid bit as the high-order bit of the cache tag and extend the address tag by adding a high-order “1” bit. Now the values are equal only if the tags match and the valid bit is a 1.

7.11 The miss penalty is the time to transfer one block from main memory to the cache. Assume that it takes one clock cycle to send the address to the main memory.

- a. Configuration (a) requires 16 main memory accesses to retrieve a cache block and words of the block are transferred 1 at a time.

$$\text{Miss penalty} = 1 + 16 \times 10 + 16 \times 1 = 177 \text{ clock cycles.}$$

- b. Configuration (b) requires 4 main memory accesses to retrieve a cache block and words of the block are transferred 4 at a time.

$$\text{Miss penalty} = 1 + 4 \times 10 + 4 \times 1 = 45 \text{ clock cycles.}$$

- c. Configuration (c) requires 4 main memory accesses to retrieve a cache block and words of the block are transferred 1 at a time.

$$\text{Miss penalty} = 1 + 4 \times 10 + 16 \times 1 = 57 \text{ clock cycles.}$$

7.12 Effective CPI = Base CPI + Miss rate per instruction \times Miss penalty.

For memory configurations (a), (b), and (c):

- a. Effective CPI = $1.2 + 0.005 \times 177 = 2.085$ clocks/instruction.

- b. Effective CPI = $1.2 + 0.005 \times 45 = 1.425$ clocks/instruction.

- c. Effective CPI = $1.2 + 0.005 \times 57 = 1.485$ clocks/instruction.

Now

$$\text{Speedup of A over B} = \text{Execution time B} / \text{Execution time A,}$$

and

$$\text{Execution time} = \text{Number of instructions} \times \text{CPI} \times \text{Clock cycle time.}$$

We have the same CPU running the same software on the various memory configurations, so the number of instructions and the clock cycle time are fixed. Thus, speed can be compared by comparing CPI.

$$\text{Speedup of configuration (b) over configuration (a)} = 2.085/1.425 = 1.46.$$

$$\text{Speedup of configuration (b) over configuration (c)} = 1.485/1.425 = 1.04.$$

7.13 The shortest reference string will have 4 misses for C1 and 3 misses for C2; this leads to 32 versus 33 miss cycles. The following reference string will do: 0, 4, 8, 11.

7.14 For C2 to have more misses, we must have a situation where a block is replaced in C2 and then another reference occurs to a different word (in C1) that was replaced. This has to happen more than once. Here's one example string:

Addresses	Cache 1	Cache 2
0	Miss	Miss
1	Miss	Hit
16	Miss	Miss
1	Hit	Miss
16	Hit	Miss
Total misses	3	4

7.15 $AMAT = \text{Hit time} + \text{Miss time} \times \text{Miss penalty}$.

$$AMAT = 2 \text{ ns} + (20 \times 2 \text{ ns}) \times 0.05 = 4 \text{ ns}.$$

7.16 $AMAT = (1.2 \times 2 \text{ ns}) + (20 \times 2 \text{ ns} \times 0.03) = 2.4 \text{ ns} + 1.2 \text{ ns} = 3.6 \text{ ns}$. Yes, this is a good choice.

7.17 $\text{Execution time} = \text{Clock cycle} \times \text{IC} \times (\text{CPI} + \text{Cache miss cycles per instruction})$

$$\text{Execution time}_{\text{original}} = \text{Clock cycle} \times \text{IC} \times (\text{CPI} + \text{Cache miss cycles per instruction})$$

$$\text{Execution time}_{\text{original}} = 2 \times \text{IC} \times (2 + 1.5 \times 20 \times 0.05) = 7 \text{ IC}$$

$$\text{Execution time}_{\text{new}} = 2.4 \times \text{IC} \times (2 + 1.5 \times 20 \times 0.03) = 6.96 \text{ IC}$$

Hence doubling the cache size to improve miss rate at the expense of stretching the clock cycle results in essentially no net gain.

7.18 The largest direct-mapped cache with one-word blocks would be 256 KB in size. The address breakdown is 14 bits for tag, 16 bits for index, 0 bits for block offset, and 2 bits for byte offset. A total of 12 chips will be required—4 for overhead.

7.19 If the block size is four words, we can build a 512-KB cache. The address breakdown is 13 bits for tag, 15 bits for index, 2 bits for block offset, and 2 bits for byte offset. A total of 18 chips will be required—2 for overhead.

7.20

Reference	Hit or miss
1	Miss
4	Miss
8	Miss
5	Miss
20	Miss
17	Miss
19	Miss
56	Miss
9	Miss
11	Miss
4	Hit
43	Miss
5	Hit
6	Miss
9	Hit
17	Hit

Here is the final state of the cache with LRU order shown right to left:

Block #	Element #1 address	Element #1 address
0	56	8
1	17	9
2		
3	43	11
4	4	20
5	5	
6	6	
7		

7.21

Reference	Hit or miss
1	Miss
4	Miss
8	Miss
5	Miss
20	Miss
17	Miss
19	Miss
56	Miss
9	Miss
11	Miss
4	Hit
43	Miss
5	Hit
6	Miss
9	Hit
17	Hit

Here is the final state of the cache:

Block #	Address
0	17
1	9
2	6
3	5
4	43
5	4
6	11
7	56
8	19
9	20
10	8
11	1
12	
13	
14	
15	

Most recently used

Least recently used

7.22

Reference	Hit or miss
1	Miss
4	Miss
8	Miss
5	Hit
20	Miss
17	Miss
19	Hit
56	Miss
9	Miss
11	Hit
4	Miss
43	Miss
5	Hit
6	Hit
9	Hit
17	Miss

Here is the final state of the cache LRU order (left to right):

Address	Address	Address	Address
40	4	8	16

7.23 Two principles apply to this cache behavior problem. First, a two-way set-associative cache of the same size as a direct-mapped cache has half the number of sets. Second, LRU replacement can behave pessimally (as poorly as possible) for access patterns that cycle through a sequence of addresses that reference more blocks than will fit in a set managed by LRU replacement.

Consider three addresses—call them A, B, C—that all map to the same set in the two-way set-associative cache, but to two different sets in the direct-mapped cache. Without loss of generality, let A map to one set in the direct-mapped cache and B and C map to another set. Let the access pattern be A B C A B C A . . . and so on. The direct-mapped cache will then have miss, miss, miss, hit, miss, miss, hit, . . . , and so on. With LRU replacement, the block at address C will replace the block at the address A in the two-way set-associative cache just in time for the A to be referenced again. Thus, the two-way set-associative cache will miss on every reference as this access pattern repeats.

7.24

Address size:	k bits
Cache size:	S bytes/cache
Block size:	$B = 2^b$ bytes/block
Associativity:	A blocks/set

Number of sets in the cache:

$$\begin{aligned} \text{Sets/cache} &= \frac{(\text{Bytes/cache})}{(\text{Blocks/set}) \times (\text{Bytes/block})} \\ &= \frac{S}{AB} \end{aligned}$$

Number of address bits needed to index a particular set of the cache:

$$\begin{aligned}\text{Cache set index bits} &= \log_2 (\text{Sets/cache}) \\ &= \log_2 \left(\frac{S}{AB} \right) \\ &= \log_2 \left(\frac{S}{A} \right) - b\end{aligned}$$

Number of two-input XOR gates needed for a tag address match for an entire set:

Tag address bits/block = (Total address bits) – (Cache set index bits) – (Block offset bits)

$$\begin{aligned}&= k - \left(\log_2 \left(\frac{S}{A} \right) - b \right) - b \\ &= k - \log_2 \left(\frac{S}{A} \right)\end{aligned}$$

XOR gates/set = (Blocks/set) × (Tag address bits/block) × (1 XOR gate/bit)

$$= A \left(k - \log_2 \left(\frac{S}{A} \right) \right)$$

Number of bits in tag memory for the cache:

Bits in tag memory/cache = (Tag address bits/block) × (Sets/cache) × (Blocks/set)

$$\begin{aligned}&= \left(k - \log_2 \left(\frac{S}{A} \right) \right) \frac{S}{AB} A \\ &= \frac{S}{B} \left(k - \log_2 \left(\frac{S}{A} \right) \right)\end{aligned}$$

7.25 Fully associative cache of 3K words? Yes. A fully associative cache has only one set, so no index is used to access the cache. Instead, we need to compare the tag with every cache location (3K of them).

Set-associative cache of 3K words? Yes. Implement a three-way set-associative cache. This means that there will be 1024 sets, which require a 10-bit index field to access.

Direct-mapped cache of 3K words? No. (Cannot be done efficiently.) To access a 3K-word direct-mapped cache would require between 11 and 12 index bits. Using 11 bits will allow us to only access 2K of the cache, while using 12 bits will cause us to map some addresses to nonexistent cache locations.

7.26 Fully associative cache of 300 words? Yes. A fully associative cache has only 1 set, so no index is used to access the cache. Instead, we need to compare the tag with every cache location (300 of them).

Set-associative cache of 300 words? Yes. Implement a 75-way set-associative cache. This means that there will be 4 sets, which require a 2-bit index field to access.

Direct-mapped cache of 300 words? No. (Cannot be done efficiently.) To access a 300-word direct-mapped cache would require between 8 and 9 index bits. Using 8 bits will allow us to only access 256 words of the cache, while using 9 bits will cause us to map some addresses to nonexistent cache locations.

7.27 Here are the cycles spent for each cache:

Cache	Miss penalty	Instruction miss cycles per instruction	Data miss cycles per data reference	Total miss cycles per instruction
C1	$6 + 1 = 7$	$4\% \times 7 = 0.28$	$8\% \times 7 = 0.56$	0.56
C2	$6 + 4 = 10$	$2\% \times 10 = 0.20$	$5\% \times 10 = 0.50$	0.45
C3	$6 + 4 = 10$	$2\% \times 10 = 0.20$	$4\% \times 10 = 0.40$	0.40

So, C3 spends the least time on misses and C1 spends the most.

7.28 Execution time = CPI \times Clock cycle \times Instruction count

$$\text{Execution time}_{C1} = 0.56 \times 2 \text{ ns} \times \text{IC} = 1.12 \times \text{IC} \times 10^{-9}$$

$$\text{Execution time}_{C2} = 0.45 \times 2 \text{ ns} \times \text{IC} = 0.9 \times \text{IC} \times 10^{-9}$$

$$\text{Execution time}_{C3} = 0.40 \times 2.4 \text{ ns} \times \text{IC} = 0.96 \times \text{IC} \times 10^{-9}$$

So C2 is the fastest and C1 is the slowest.

7.29 If direct-mapped and `stride = 132`, then we can assume without loss of generality that `array[0]` is in slot 0, along with `array[1]`, `[2]`, `[3]`. Then, slot 1 has `[4]`, `[5]`, `[6]`, `[7]`, slot 2 has `[8]`, `[9]`, `[10]`, `[11]`, and so on until slot 15 has `[60]`, `[61]`, `[62]`, `[63]` (there are 16 slots \times 16 bytes = 256 bytes in the cache). Then wrapping around, we find also that slot 0 has `[64]`, `[65]`, `[66]`, `[67]` and also `[128]`, `[129]`, `[130]`, `[131]`, etc. Thus if we look at `array[0]`, `array[132]`, we are looking at two different slots. After filling these entries, there will be no misses. Thus the expected miss rate is about 0. If the stride equals 131, we are looking at slot 0 and slot 0 again. Each reference will bounce out the other, and we will have 100% misses. If the cache is two-way set-associative, even if two accesses are in the same cache line they can coexist, so the miss rate will be 0. Alternative explanation: 132 in binary is 10000100, and clearly when this is added to the word address of `array[0]`, bit 2 will change and thus the slots are different. 131 in binary is 10000011, and in this case the last two bits are used for the block offset within the cache. The addition will not change the index, so the two entries will map to the same location.

7.30 We can build a 384-KB cache. The address breakdown is 15 bits for tag, 15 bits for set index, 0 bits for block offset, and 2 bits for byte offset. A total of 18 chips will be required; 6 for overhead.

7.31 No solution provided.

7.32 The total size is equal to the number of entries times the size of each entry. The number of entries is equal to the number of pages in the virtual address, which is

$$\frac{2^{40} \text{ bytes}}{16 \text{ KB}} = \frac{2^{40} \text{ bytes}}{2^4 2^{10} \text{ bytes}} = 2^{26}$$

The width of each entry is $4 + 3 \text{ bits} = 40 \text{ bits} = 8 \text{ bytes}$. Thus the page table contains 2^{29} bytes or 512 MB!

7.33 No solution provided.

7.34 No solution provided.

7.35–7.36 Pages are 4-KB in size and each entry uses 32 bits, so we get 1K worth of page table entries in a page. Each of these entries points to a physical 4-KB page, making it possible to address $2^{10} \times 2^{12} = 2^{22}$ bytes = 4 MB of memory. But only half of these are valid, so 2 MB of our virtual address space would be in physical memory. If there are 1K worth of entries per page table, the page table offset will occupy 10 bits and the page table number also 10 bits. Thus we only need 4 KB to store the first-level page table as well.

7.37 No solution provided.

7.38 Rewriting the program will likely reduce compulsory as well as capacity misses (if you are lucky, maybe they will conflict). Increasing the clock rate will have no change. Increasing the associativity should reduce conflict.

7.39 No solution provided.

7.40 No solution provided.

7.41 No solution provided.

7.42 No solution provided.

7.43 No solution provided.

7.44 No solution provided.

7.45 This optimization takes advantage of spatial locality. This way we update all of the entries in a block before moving to another block.

7.46 No solution provided.

7.47 No solution provided.

7.48 No solution provided.

8

Solutions

8.1 Each transaction requires 50,000 instructions and 5 I/O operations. The CPU can handle transactions at a maximum rate of 50M/50K or 1000 transactions per second. The I/O limit for system A is 200 TPS and for system B it's 150 TPS. These are the limiting rates.

8.2 For system A, $20 \text{ ms}/\text{IO} \times 5 \text{ IO}/\text{transaction} \times n \text{ transactions}/10 = 1000 \text{ ms}$, $n = 100 \text{ TPS}$.

Assume system B operates at greater than 100 TPS (and less than 500 I/O operations per second). Let n be the number over 100:

$$(18 \times 5 \times 100 + n \times 5 \times 25)/10 = 1000 \text{ ms. } n = 8 \text{ or } 108 \text{ TPS.}$$

This solution ignores the fact that the first set of transactions that can be performed simultaneously numbers only 9. After the first 9 transactions, subsequent transactions can be processed in batches of 10, comprising one transaction dependent on a preceding transaction and a new set of 9 independent transactions.

8.3 After reading sector 7, a seek is necessary to get to the track with sector 8 on it. This will take some time (on the order of a millisecond, typically), during which the disk will continue to revolve under the head assembly. Thus, in the version where sector 8 is in the same angular position as sector 0, sector 8 will have already revolved past the head by the time the seek is completed and some large fraction of an additional revolution time will be needed to wait for it to come back again. By skewing the sectors so that sector 8 starts later on the second track, the seek will have time to complete, and then the sector will soon thereafter appear under the head without the additional revolution.

8.4 For ATM to be twice as fast, we would need

$$15 + 6 + 200 + 241 + \text{Transmission time}_{\text{Ethernet}} = 2 \times (50 + 6 + 207 + 360 + \text{Transmission time}_{\text{ATM}})$$

$$462 + X/1.125 = 2(623 + X/10), \text{ so } X = 1138 \text{ bytes.}$$

8.5 $100 / (3 \times 10^8 \times .5) = .67 \text{ microseconds}$ and $5000 \times 10^3 / (3 \times 10^8 \times .5) = .034 \text{ seconds}$.

8.6 $.67 \times 10^{-6} \times 5 \times 10^6 = 3.35 \text{ bytes}$ for the 100-meter network, and $.034 \times 5 \times 10^6 = .17 \text{ MB}$ for the 5000-km network.

8.7 $4 \text{ KHz} = 4 \times 10^3 \text{ samples/sec} \times 2 \text{ bytes/sample} = 8 \times 10^3 \text{ bytes/sec} \times 100 \text{ conversations} = 8 \times 10^5 \text{ bytes/sec}$. Transmission time per packet is $1 \text{ KB}/1 \text{ MB/sec} = 1 \text{ millisecond}$ plus a 350-microsecond latency for a total of 0.00135 seconds. The time to transmit all 800 packets collected from one second of monitoring is $800 \times 0.00135 = 1.08 \text{ seconds}$. Thus, the chosen network with its given latency does not have sufficient effective bandwidth for the task.

8.8 We determine an average disk access time of $8 \text{ ms} + 4.2 \text{ ms} + .2 \text{ ms} + 2 \text{ ms} = 14.4 \text{ ms}$. Since each block processed involves two accesses (read and write), the disk component of the time is 28.8 ms per block processed. The (non-overlapped) computation takes 20 million cycles at 400 MHz, or another 50 ms. Thus, the total time to process one block is 78.8 ms, and the number of blocks processed per second is simply $1/0.0788 = 12.7$.

8.9 For Ethernet, Latency = 462 + Transmission time. For ATM we have Latency = 623 + Transmission time. For Ethernet, transmission times are 8.9 and 133.3; for ATM, 1 and 15. Thus, Ethernet total = 1066 and ATM = 1262. For this application, Ethernet is better due to the smaller latencies (all units are microseconds).

8.10 For four-word block transfers, the bus bandwidth was 71.11 MB/sec. For 16-word block transfers, the bus bandwidth was 224.56 MB/sec. The disk drive has a transfer rate of 5 MB/sec. Thus for four-word blocks we could sustain $71/5 = 14$ simultaneous disk transfers, and for 16-word blocks we could sustain $224 / 5 = 44$ simultaneous disk transfers. The number of simultaneous disk transfers is inherently an integer and we want the sustainable value. Thus we take the floor of the quotient of bus bandwidth divided by disk transfer rate.

8.11 For the four-word block transfers, each block now takes

1. 1 cycle to send an address to memory
2. $150 \text{ ns} / 5 \text{ ns} = 30$ cycles to read memory
3. 2 cycles to send the data
4. 2 idle cycles between transfers

This is a total of 35 cycles, so the total transfer takes $35 \times 64 = 2240$ cycles. Modifying the calculations in the example, we have a latency of 11,200 ns, 5.71M transactions/second, and a bus bandwidth of 91.43 MB/sec.

For the 16-word block transfers, each block now takes

1. 1 cycle to send an address to memory
2. 150 ns or 30 cycles to read memory
3. 2 cycles to send the data
4. 4 idle cycles between transfers, during which the read of the next block is completed

Each of the three remaining four-word blocks requires repeating the last two steps. This is a total of $1 + 30 + 4 \times (2 + 4) = 55$ cycles, so the transfer takes $55 \times 16 = 880$ cycles. We now have a latency of 4400 ns, 3.636M transactions/second, and a bus bandwidth of 232.73 MB/sec.

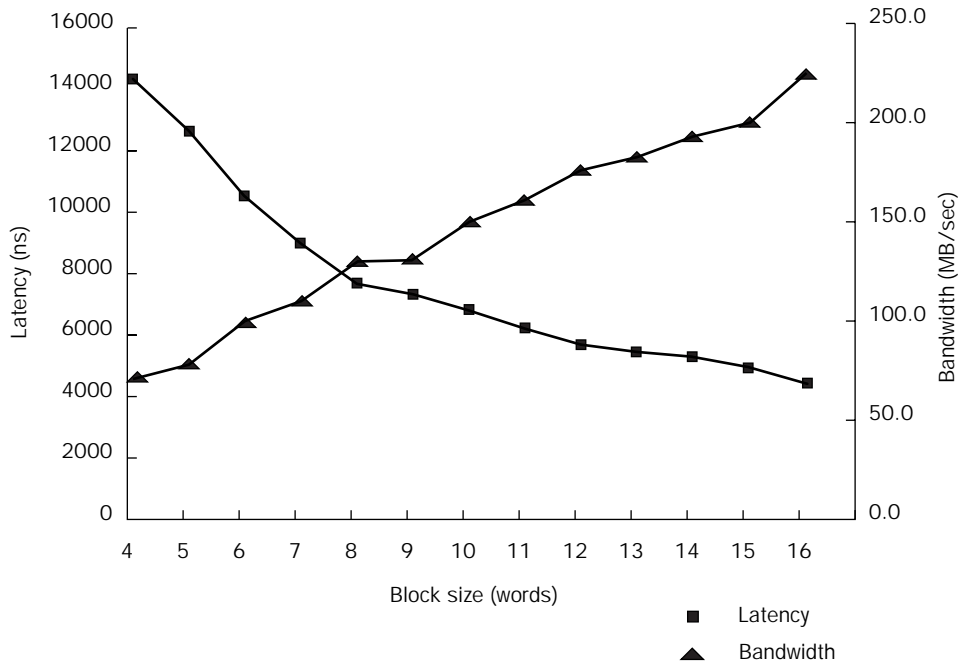
Note that the bandwidth for the larger block size is only 2.54 times higher given the new read times. This is because the 30 ns for subsequent reads results in fewer opportunities for overlap, and the larger block size performs (relatively) worse in this situation.

8.12 The key advantage would be that a single transaction takes only 45 cycles, as compared with 57 cycles for the larger block size. If because of poor locality we were not able to make use of the extra data brought in, it might make sense to go with a smaller block size. Said again, the example assumes we want to access 256 words of data, and clearly larger block sizes will be better. (If it could support it, we'd like to do a single 256-word transaction!)

8.13 Assume that only the four-word reads described in the example are provided by the memory system, even if fewer than four words remain when transferring a block. Then,

	Block size (words)												
	4	5	6	7	8	9	10	11	12	13	14	15	16
Number of four-word transfers to send the block	1	2	2	2	2	3	3	3	3	4	4	4	4
Time to send address to memory (bus cycles)	1	1	1	1	1	1	1	1	1	1	1	1	1
Time to read first four words in memory (bus cycles)	40	40	40	40	40	40	40	40	40	40	40	40	40
Block transfer time, at 2 transfer bus cycles and 2 idle bus cycles per four-word transfer (bus cycles)	4	8	8	8	8	12	12	12	12	16	16	16	16
Total time to transfer one block (bus cycles)	45	49	49	49	49	53	53	53	53	57	57	57	57
Number of bus transactions to read 256 words using the given block size	64	52	43	37	32	29	26	24	22	20	19	18	16
Time for 256-word transfer (bus cycles)	2880	2548	2107	1813	1568	1537	1378	1272	1166	1140	1083	1026	912
Latency (ns)	14400	12740	10535	9065	7840	7685	6890	6360	5830	5700	5415	5130	4560
Number of bus transactions (millions per second)	4.444	4.082	4.082	4.082	4.082	3.774	3.774	3.774	3.774	3.509	3.509	3.509	3.509
Bandwidth (MB/sec)	71.1	80.4	97.2	113.0	130.6	133.2	148.6	161.0	175.6	179.6	189.1	199.6	224.6

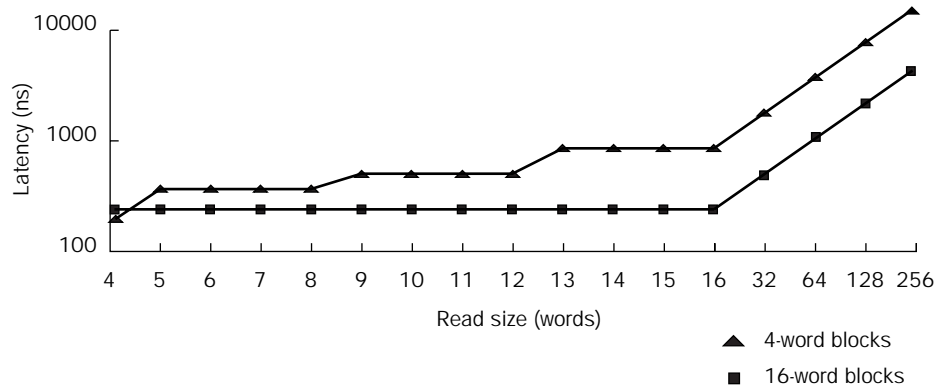
The following graph plots latency and bandwidth versus block size.



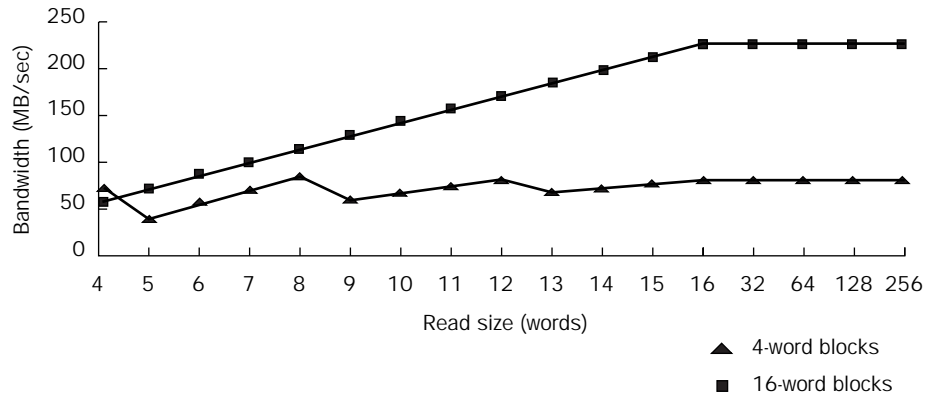
8.14 From the example, a four-word transfer takes 45 bus cycles and a 16-word block transfer takes 57 bus cycles. Then,

	Read size (words)																
	4	5	6	7	8	9	10	11	12	13	14	15	16	32	64	128	256
Number of four-word transfers to send the data	1	2	2	2	2	3	3	3	3	4	4	4	4	8	16	32	64
Number of 16-word transfers to send the data	1	1	1	1	1	1	1	1	1	1	1	1	1	2	4	8	16
Total read time using four-word blocks (bus cycles)	45	90	90	90	90	135	135	135	135	180	180	180	180	360	720	1440	2880
Total read time using 16-word blocks (bus cycles)	57	57	57	57	57	57	57	57	57	57	57	57	57	114	228	456	912
Latency using four-word blocks (ns)	225	450	450	450	450	675	675	675	675	900	900	900	900	1800	3600	7200	14400
Latency using 16-word blocks (ns)	285	285	285	285	285	285	285	285	285	285	285	285	285	570	1140	2280	4560
Bandwidth using four-word blocks (MB/sec)	71.1	44.4	53.3	62.2	71.1	53.3	59.3	65.2	71.1	57.8	62.2	66.7	71.1	71.1	71.1	71.1	71.1
Bandwidth using 16-word blocks (MB/sec)	56.1	70.2	84.2	98.2	112.3	126.3	140.4	154.4	168.4	182.5	196.5	210.5	224.6	224.6	224.6	224.6	224.6

The following graph plots read latency with 4-word and 16-word blocks.



The following graph plots bandwidth with 4-word and 16-word blocks.



8.15

For four-word blocks:

Send address and first word simultaneously = 1 clock
 Time until first write occur = 40 clocks
 Time to send remaining three words over 32-bit bus = 3 clocks
 Required bus idle time = 2 clocks
 Total time = 46 clocks

Latency = 64 four-word blocks at 46 cycles per block = 2944 clocks = 14720 ns.

Bandwidth = $(256 \times 4 \text{ bytes}) / 14720 \text{ ns} = 69.57 \text{ MB/sec}$.

For eight-word blocks:

Send address and first word simultaneously = 1 clock
 Time until first write occurs = 40 clocks
 Time to send remaining seven words over 32-bit bus = 7 clocks
 Required bus idle time (two idle periods) = 4 clocks
 Total time = 52 clocks

Latency = 32 8-word blocks at 52 cycles per block = 1664 clocks = 8320 ns.

Bandwidth = $(256 \times 4 \text{ bytes}) / 8320 \text{ ns} = 123.08 \text{ MB/sec}$.

In neither case does the 32-bit address/32-bit data bus outperform the 64-bit combined bus design. For smaller blocks there could be an advantage if the overhead of a fixed four-word block bus cycle could be avoided.

8.16 For a 16-word read from memory, there will be four sends from the four-word-wide memory over the four-word-wide bus. Transactions involving more than one send over the bus to satisfy one request are typically called *burst transactions*.

For burst transactions, some way must be provided to count the number of sends so that the end of the burst will be known to all on the bus. We don't want another device trying to access memory in a way that interferes with an ongoing burst transfer. The common way to do this is to have an additional bus control signal, called BurstReq or Burst Request, that is asserted for the duration of the burst. This signal is unlike the ReadReq signal of Figure 8.10, which is asserted only long enough to start a single transfer. One of the devices can incorporate the counter necessary to track when BurstReq should be deasserted, but both devices party to the burst transfer must be designed to handle the specific burst (four words, eight words, or other amount) desired. For our bus, if BurstReq is not asserted when ReadReq signals the start of a transaction, then the hardware will know that a single send from memory is to be done.

So the solution for the 16-word transfer is as follows: The steps in the protocol begin immediately after the device signals a burst transfer request to the memory by raising ReadReq and Burst_Request and putting the address on the Data lines.

1. When memory sees the ReadReq and BurstReq lines, it reads the address of the start of the 16-word block and raises Ack to indicate it has been seen.
2. I/O device sees the Ack line high and releases the ReadReq and Data lines, but it keeps BurstReq raised.
3. Memory sees that ReadReq is low and drops the Ack line to acknowledge the ReadReq signal.
4. This step starts when BurstReq is high, the Ack line is low, and the memory has the next four data words ready. Memory places the next four data words in answer to the read request on the Data lines and raises DataRdy.
5. The I/O device sees DataRdy, reads the data from the bus, and signals that it has the data by raising Ack.
6. The memory sees the Ack signal, drops DataRdy, and releases the Data lines.
7. After the I/O device sees DataRdy go low, it drops the Ack line but continues to assert BurstReq if more data remains to be sent to signal that it is ready for the next four words. Step 4 will be next if BurstReq is high.
8. If the last four words of the 16-word block have been sent, the I/O device drops BurstReq, which indicates that the burst transmission is complete.

With handshakes taking 20 ns and memory access taking 60 ns, a burst transfer will be of the following durations:

Step 1 20 ns (memory receives the address at the end of this step; data goes on the bus at the beginning of step 5).

Steps 2, 3, 4 Maximum (3×20 ns, 60 ns) = 60 ns.

Steps 5, 6, 7, 4 Maximum (4×20 ns, 60 ns) = 80 ns (looping to read and then send the next four words; memory read latency completely hidden by handshaking time).

Steps 5, 6, 7, 4 Maximum (4×20 ns, 60 ns) = 80 ns (looping to read and then send the next four words; memory read latency completely hidden by handshaking time).

Steps 5, 6, 7, 4 Maximum (4×20 ns, 60 ns) = 80 ns (looping to read and then send the next four words; memory read latency completely hidden by handshaking time).

End of burst transfer.

Thus, the total time to perform the transfer is 320 ns, and the maximum bandwidth is

$$(16 \text{ words} \times 4 \text{ bytes}) / 320 \text{ ns} = 200 \text{ MB/sec.}$$

It is a bit difficult to compare this result to that in the example on page 665 because the example uses memory with a 200-ns access instead of 60 ns. If the slower memory were used with the asynchronous bus, then the total time for the burst transfer would increase to 820 ns, and the bandwidth would be

$$(16 \text{ words} \times 4 \text{ bytes}) / 820 \text{ ns} = 78 \text{ MB/sec.}$$

The synchronous bus in the example on page 665 needs 57 bus cycles at 5 ns per cycle to move a 16-word block. This is 285 ns, for a bandwidth of

$$(16 \text{ words} \times 4 \text{ bytes}) / 285 \text{ ns} = 225 \text{ MB/sec.}$$

8.17 The new read size results in a maximum I/O rate of the bus = $(100 \times 10^6) / (4 \times 10^3) = 25,000$ I/Os per second. In this situation, the CPU is now the bottleneck and we can configure the rest of the system to perform at the level dictated by the CPU of 2000 I/Os per second. Time per I/O at disk now becomes 10.8 ms, which means each disk can complete $1000 / 10.8 = 92.6$ I/Os per second. To saturate the CPU requires 2000 I/Os per second or 22 disks, which will require four controllers.

8.18 First, check that neither the memory nor the I/O bus is a bottleneck. Sustainable bandwidth into memory is 4 bytes per 2 clocks = 400 MB/sec. The I/O bus can sustain 10 MB/sec. Both of these are faster than the disk bandwidth of 5 MB/sec, so when the disk transfer is in progress there will be negligible additional time needed to pass the data through the I/O bus and to write it into the memory. Thus we ignore this time and focus on the time needed by the DMA controller and the disk. This will take 1 ms to initiate, 12 ms to seek, $16 \text{ KB} / 5 \text{ MB} = 3.2$ ms to transfer: total = 16.2 ms.

8.19 The processor-memory bus takes 8 cycles to accept 4 words, or 2 bytes/clock cycle. This is a bandwidth of 400 MB/sec. Thus we need $400 / 5 = 80$ disks, and because all 80 are transmitting, we need $400 / 10 = 40$ I/O buses.

8.20 We can't saturate the memory (we'd need 32 more disks and 34 more I/O buses). At most 12 disks can transfer at a time, because we have 6 I/O buses and the I/O bus is at 10 MB/sec and the disks at 5 MB/sec. Another limitation is the seek and rotational latency. For this size transfer, the overhead time is 13 ms (1 + 12) and the transfer time is $4 \text{ KB} / 5 \text{ MB} = .8$ ms. Thus the disks are actually spending most of their time finding the data, not transferring it. So each disk is only transferring $.8 / 13.8 = 5.8\%$ of the time. Every I/O is serving 8 disks but is not saturated, because the 8 disks act like .46 (less than one disk) constantly transferring. A disk constantly transferring can perform $5 \text{ MB} / 4 \text{ KB} = 1250$ transfers each second. Thus the maximum I/O rate is 6 (controllers) $\times .46$ disks $\times 1250 = 3450$ I/Os per second. This is an I/O bandwidth of roughly $3450 \times 4 \text{ KB} = 13.8 \text{ MB/sec}$.

8.21 To saturate an I/O bus we need to have 2 simultaneous transfers from 8 disks. This means that $2 = 8 \times (T / (13 + T))$ where T is the transfer time. So, $T = 4.33$ ms. This corresponds to 21.67-KB reads, so the blocks must be 32 KB. A disk constantly transferring can perform $5 \text{ MB} / 32 \text{ KB} = 156.25$ transfers each second. With this block size we can perform 2 transfers per I/O bus for a total of 312.5 transfers per bus per second. We have 6 buses, so we can perform 1875 transfers per second. This is a bandwidth of $32 \text{ KB} \times 1875 = 60 \text{ MB/sec}$.

8.22 How many cycles to read or write 8 words? Using the current example (64-bit bus with 128-bit memory bus, $1 + 40 + 2 \times (2 + 2) = 49$ cycles. The average miss penalty is $.4 \times 49 + 49 = 68.6$ cycles. Miss cycles per instruction = $68.6 \times .005 = 3.43$. If we up the block size to 16, we get 57 cycles for 16 words, which makes $79.8 \times .03 = 2.39$ miss cycles per instruction.

8.23 Some simplifying assumptions are

- a fixed overhead for initiating and ending the DMA in units of clock cycles. This ignores memory hierarchy misses adding to the time.
- disk transfers take the same time as the time for the average size transfer, but the average transfer size may not well represent the distribution of actual transfer sizes.
- real disks will not be transferring 100% of the time; far from it.

8.24 No solution provided.

8.25 No solution provided.

8.26 No solution provided.

8.27 No solution provided.

8.28 No solution provided.

8.29 Each disk access requires 1 ms of overhead + 6 ms of seek.

For the 4-KB access (4 sectors):

- Single disk requires 4.17 ms + 0.52 ms (access time) + 7 ms = 11.7 ms.
- Disk array requires 4.17 ms + 0.13 ms (access time) + 7 ms = 11.3 ms.

For the 16-KB access (16 sectors):

- Single disk requires 4.17 ms + 2.08 ms (access time) + 7 ms = 13.25 ms.
- Disk array requires 4.17 ms + 0.52 ms (access time) + 7 ms = 11.7 ms.

Here are the total times and throughput in I/Os per second:

- Single disk requires $(11.7+13.25)/2 = 12.48$ ms and can do 80.2 I/Os per second.
- Disk array requires $(11.3+11.7)/2 = 11.5$ ms and can do 87.0 I/Os per second.

For the solution to the challenge: The actual average rotational latency if we assume that the disk head becomes ready to read from a track at a random location on that track is as follows.

For 1-sector reads (done by the disk array when reading a 4-KB block), there is a $1/64$ chance that the disk head lands in the sector to read. The disk head must wait on average for the disk to rotate $63.5/64$ of a complete rotation to reach the start of the sector. There is a $63/64$ chance that the head starts in a sector other than the one to read and must wait the rotational distance to the start of the sector to read, which is the number of whole sectors between the start point and the sector to read (ranges from 0 to 62) plus, on average, one half of the sector where the head started. So the average rotational latency in number of sectors when reading 1 KB is

$$\frac{63.5}{64} + \frac{1}{64} \sum_{i=0}^{62} (i + 0.5) = \frac{1}{64} (63.5 + 1953 + 31.5) = 32 \text{ sectors.}$$

The time to rotate past one sector is $60/7200 \times 64 = 0.13$ ms, so the actual average latency is $32 \times 0.13 = 4.16$ ms.

For 4-sector reads (single disk reading 4 KB and disk array reading 16 KB) there is a $1/16$ chance of starting within the 4 KB of sequential data, which requires a full rotational latency plus on average half of a sector ($1/8$ of the 4-sector region) to complete. (Note that sectors comprising the 4 KB may be read out of order, but we assume that partial sectors do not contribute to the 4-sector read, i.e., reading starts at the first sector boundary encountered.) There is a $15/16$ chance of starting in a sector outside of the read area, and the latency then is the whole number of sectors between the head start position and the consecutive sectors to read (ranging from 0 to 14), plus one half on average of a 4-sector segment. Thus,

$$\frac{1}{16} \left(1 + \frac{1}{8}\right) + \frac{1}{16} \sum_{i=0}^{14} (i + 0.5) = \frac{1}{16} (1 + 0.125 + 105 + 7.5) = 7.1 \text{ 4-sector segments.}$$

The time to rotate past a 4-sector segment is $60/7200 \times 16 = 0.52$ ms, so the actual average latency is $7.1 \times 0.52 = 3.69$ ms.

For 16-sector reads (single disk reading 16 KB) we have, by similar reasoning to the 4-sector read case,

$$\frac{1}{4} \left(1 + \frac{1}{32} \right) + \frac{1}{4} \sum_{i=0}^2 (i + 0.5) = \frac{1}{4} (1 + 0.03125 + 3 + 1.5) = 1.383 \text{ 16-sector segments.}$$

The time to rotate past a 16-sector segment is $60/7200 \times 4 = 2.08$ ms, so the actual average latency is $1.383 \times 2.08 = 2.88$ ms.

The rotational latency decreases as the area to be read occupies more of the track, provided the sectors can be read out of order. Reordering the sector data once it is read can be readily and rapidly accomplished by appropriate disk controller electronics.

8.30 The average read is $(4 + 16)/2 = 10$ KB. Thus, the bandwidths are

Single disk: $80.2 \times 10 \text{ KB} = 802 \text{ KB/second}$.

Disk array: $87.0 \times 10 \text{ KB} = 870 \text{ KB/second}$.

9

Solutions

9.1 No solution provided.

9.2 The short answer is that x is obviously always a 1, and y can either be 1, 2, or 3. In a load-store architecture the code might look like the following:

Processor 1

```
load X into a register
increment register
store register back to X
load Y into a register
add two registers to register
store register back to Y
```

Processor 2

```
load X into a register
increment register
store register back to Y
```

For considering the possible interleavings, only the loads/stores are really of interest. There are four activities of interest in process 1 and two in process 2. There are 15 possible interleavings, which result in the following:

111122: $x = 1, y = 2$

111212: $x = 1, y = 2$

111221: $x = 1, y = 1$

112112: $x = 1, y = 2$

112121: $x = 1, y = 1$

112211: $x = 1, y = 3$

121112: $x = 1, y = 1$

121121: $x = 1, y = 1$

121211: $x = 1, y = 2$

122111: $x = 1, y = 2$

211112: $x = 1, y = 1$

211121: $x = 1, y = 1$

211211: $x = 1, y = 2$

212111: $x = 1, y = 2$

221111: $x = 1, y = 2$

9.3 No solution provided.

9.4 No solution provided.

9.5 Write-back cache with write-update cache coherency and one-word blocks. All words in cache are initially clean. Assume four-byte words and byte addressing.

Step	Action	Comment
1	P1 writes 100	One bus transfer to move the word at 100 from P1 to P2 cache; no transfer from P2 cache to main memory because all blocks are clean.
2	P2 writes 104	One bus transfer to move the word at 104 from P2 to P1 cache; no transfer from P1 cache to main memory because the transferred block maps to a clean location in P1 cache.
3	P1 reads 100	No bus transfer; word read from P1 cache.
4	P2 reads 104	No bus transfer; word read from P2 cache.

Total bus transactions = 2.

9.6 Write-back cache with write-update cache coherency and four-word blocks. All words in cache are initially clean. Assume four-byte words and byte addressing. Assume that the bus moves one word at a time. Addresses 100 and 104 are contained in the one block starting at address 96.

Step	Action	Comment
1	P1 writes 100	Four bus transfers to move the block at 96, which contains the word at 100 from P1 to P2 cache; no transfer from P2 cache to main memory because all P2 cache blocks are clean; however, the P1 cache block holding 100 is now dirty.
2	P2 writes 104	Eight bus transfers to move the block at 96, which contains the word at 104, from P2 to P1 cache, and to move the dirty block in P1 cache to main memory.
3	P1 reads 100	No bus transfer; P1 cache has current value.
4	P2 reads 104	No bus transfer; P2 cache has current value.

Total bus transactions = 12.

9.7 No solution provided.

9.8 No solution provided.

9.9 No solution provided.

9.10 No solution provided.

9.11 No solution provided.

9.12 No solution provided.

9.13 No solution provided.

9.14 No solution provided.

A

Solutions

A.1 No solution provided.

A.2 No solution provided.

A.3 No solution provided.

A.4 No solution provided.

A.5 No solution provided.

A.6

```
main:
    li    $a0, 0
loop:
    li    $v0, 5
    syscall
    add   $a0, $a0, $v0
    bne   $v0, $zero, loop
    li    $v0, 1
    syscall
    li    $v0, 10
    syscall
```

A.7

```
main:
    li    $v0, 5
    syscall
    move  $a0, $v0
    li    $v0, 5
    syscall
    move  $a1, $v0
    li    $v0, 5
    syscall
    move  $a2, $v0
    slt   $t0, $a0, $a1
    bne   $t0, $zero, a0_or_a2_smallest
    slt   $t0, $a1, $a2
    bne   $t0, $zero, a1_smallest
a2_smallest:
    add   $a0, $a0, $a1
    j     print_it
a1_smallest:
    add   $a0, $a0, $a2
    j     print_it
a0_or_a2_smallest:
    slt   $t0, $a0, $a2
    beq   $t0, $zero, a2_smallest
    add   $a0, $a2, $a1
print_it:
```

```

        li      $v0, 1
        syscall
        li      $v0, 10
        syscall

```

A.8 We start by prompting for an integer and validating the input. Once the integer is in a register, we repeatedly (and successively) divide it by 10, keeping the remainders in an array. These remainders are nothing but the integer's digits, right to left. We then read these digits left to right and use each as an index into the "names" array and output the corresponding names. To that end, we either fix the length of the elements of the "name" array, or we keep them variable and define a second array of pointers whose elements are the addresses of the names and are, thus, of fixed length (an address is always 4 bytes). We choose the former approach:

```

        .data
msg:      .asciiz "Invalid Entry\n"
prompt:   .asciiz "Enter a positive integer ... "
names:    .asciiz "Zero "
          .space 1
          .asciiz "One "
          .space 2
          .asciiz "Two "
          .space 2
          .asciiz "Three "
          .asciiz "Four "
          .space 1
          .asciiz "Five "
          .space 1
          .asciiz "Six "
          .space 2
          .asciiz "Seven "
          .asciiz "Eight "
          .asciiz "Nine "
          .space 1
digits:   .space 10
          .globl main
          .text
          #------ Push the ret. address
main:     sw      $ra, 0($sp)
          addi   $sp, $sp, -4
          #------ Print a prompt
          addi   $v0, $zero, 4
          addi   $a0, $zero, prompt
          syscall
          #------ Input an integer into v0
          addi   $v0, $zero, 5
          syscall
          #------ Reject if non positive
          bgtz   $v0, cont
          addi   $v0, $zero, 4
          addi   $a0, $zero, msg
          syscall
          j      fini

```

```

cont:    #----- Extract its digits (right-
         to-left)
         addi   $a0, $zero, 10
         addi   $t0, $zero, 0
again:   div    $v0, $a0
         mflo   $v0          # v0 = v0 div 10
         mfhi   $t1          # t1 = v0 mod 10
         sb     $t1, digits($t0)
         beq    $v0, $zero, abort # leave if v0 became 0
         beq    $t0, $a0, abort  # or if 10 digits were extracted
         addi   $t0, $t0, 1
         j     again
abort:   #----- Print their names
         addi   $t7, $zero, 7   # t7 = 7
         lb     $t5, digits($t0)
         beq    $t5, $zero, skip # skip leading zero
next:    mult   $t5, $t7
         mflo   $t5          # t5 = the name's offset
         addi   $a0, $t5, names # a0 = the name's address
         addi   $v0, $zero, 4
         syscall
skip:    sub    $t0, $t0, 1
         bltz   $t0, fini
         lb     $t5, digits($t0)
         j     next
         #----- Pop the stack; ret to SPIM
fini:    addi   $sp, $sp, 4
         lw     $ra, 0($sp)
         jr    $ra

```

A.9 No solution provided.

A.10

```

main:
         addiu  $sp, $sp, -4
         sw    $ra, 4($sp)
         li    $v0, 4
         .data
S1:     .asciiz "Enter number of disks "
         .text
         la    $a0, S1
         syscall
         li    $v0, 5
         syscall
         move  $a0, $v0
         li    $a1, 1
         li    $a2, 2
         li    $a3, 3
         jal  hanoi
         li    $v0, 0

```



```

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

hanoi:
        beq    $a0, $zero, baseCase
        addiu  $sp, $sp, -20
        sw     $a0, 4($sp)
        sw     $a1, 8($sp)
        sw     $a2, 12($sp)
        sw     $a3, 16($sp)
        sw     $ra, 20($sp)
        addi   $a0, $a0, -1
        move   $t0, $a3
        move   $a3, $a2
        move   $a2, $t0
        jal   hanoi
        li    $v0, 4
        .data
S2:     .asciiz "Move disk "
        .text
        la    $a0, S2
        syscall
        li    $v0, 1
        lw    $a0, 4($sp)
        syscall
        li    $v0, 4
        .data
S3:     .asciiz " from peg "
        .text
        la    $a0, S3
        syscall
        li    $v0, 1
        lw    $a0, 8($sp)
        syscall
        li    $v0, 4
        .data
S4:     .asciiz " to peg "
        .text
        la    $a0, S4
        syscall
        li    $v0, 1
        lw    $a0, 12($sp)
        syscall
        li    $v0, 4
        .data
S5:     .asciiz ".\n"
        .text
        la    $a0, S5
        syscall
        lw    $a0, 4($sp)
        addi  $a0, $a0, -1

```

```
lw    $a1, 16($sp)
lw    $a2, 12($sp)
lw    $a3, 8($sp)
jal   hanoi
lw    $ra, 20($sp)
addiu $sp, $sp, 20
jr    $ra
```

```
baseCase:
jr    $ra
```

B

Solutions

B.1 This is easily shown by induction.

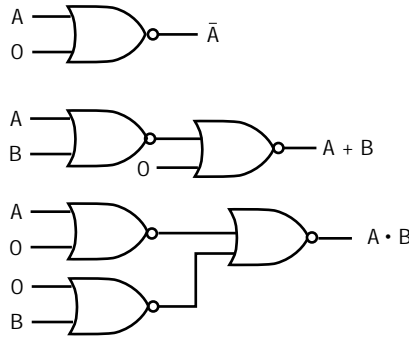
Base case: A truth table with one input clearly has two entries (2^1).

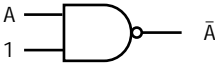
Assume a truth table with n inputs has 2^n entries.

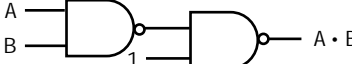
A truth table with $n + 1$ entries has 2^n entries where the $n + 1$ input is 0 and 2^n entries where the $n + 1$ input is 1, for a total of 2^{n+1} entries.

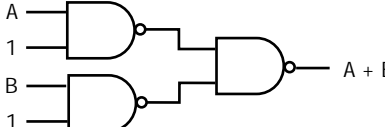
B.2 No solution provided.

B.3

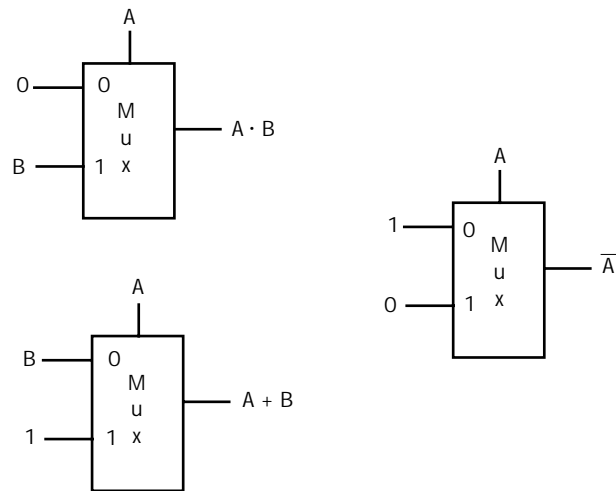


B.4 NOT function: $\bar{1} \cdot A = \bar{A}$ so 

AND function: $\overline{\bar{A} \cdot \bar{B}} = A \cdot B$ so 

OR function: $\overline{\bar{A} \cdot \bar{B}} = \bar{\bar{A}} \cdot \bar{\bar{B}} = A + B$ so 

B.5



B.6

A	B	\bar{A}	\bar{B}	$\bar{A} + \bar{B}$	$\bar{A} \cdot \bar{B}$	$\bar{A} \cdot B$	$A + \bar{B}$
0	0	1	1	1	1	1	1
0	1	1	0	0	0	1	1
1	0	0	1	0	0	1	1
1	1	0	0	0	0	0	0

B.7 Here is the first equation:

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot (\overline{A \cdot B \cdot C}).$$

Now use DeMorgan's law to rewrite the last factor:

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot (\bar{A} + \bar{B} + \bar{C})$$

Now distribute the last factor:

$$E = ((A \cdot B) \cdot (\bar{A} + \bar{B} + \bar{C})) + ((A \cdot C) \cdot (\bar{A} + \bar{B} + \bar{C})) + ((B \cdot C) \cdot (\bar{A} + \bar{B} + \bar{C}))$$

Now distribute within each term; we show one example:

$$((A \cdot B) \cdot (\bar{A} + \bar{B} + \bar{C})) = (A \cdot B \cdot \bar{A}) + (A \cdot B \cdot \bar{B}) + (A \cdot B \cdot \bar{C}) = 0 + 0 + (A \cdot B \cdot \bar{C})$$

(This is simply $A \cdot B \cdot \bar{C}$.) Thus, the equation above becomes

$$E = (A \cdot B \cdot \bar{C}) + (A \cdot \bar{B} \cdot C) + (\bar{A} \cdot B \cdot C), \text{ which is the desired result.}$$

B.8 Generalizing DeMorgan's theorems for this exercise, if $\overline{A + B} = \bar{A} \cdot \bar{B}$, then $\overline{A + B + C} = \bar{A} + \overline{(B + C)} = \bar{A} \cdot \overline{(B \cdot C)} = \bar{A} \cdot (\bar{B} \cdot \bar{C}) = \bar{A} \cdot \bar{B} \cdot \bar{C}$.

Similarly,

$$\overline{A \cdot B \cdot C} = \overline{A \cdot (B \cdot C)} = \bar{A} + \overline{B \cdot C} = \bar{A} \cdot (\bar{B} + \bar{C}) = \bar{A} + \bar{B} + \bar{C}.$$

Intuitively, DeMorgan's theorems say that (1) the negation of a sum-of-products form equals the product of the negated sums, and (2) the negation of a product-of-sums form equals the sum of the negated products. So,

$$\begin{aligned}
 E &= \overline{\overline{E}} \\
 &= \overline{(A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})} \\
 &= \overline{(A \cdot B \cdot \overline{C}) \cdot (A \cdot C \cdot \overline{B}) \cdot (B \cdot C \cdot \overline{A})}; \text{ first application of DeMorgan's theorem} \\
 &= \overline{(\overline{A} + \overline{B} + C) \cdot (\overline{A} + \overline{C} + B) \cdot (\overline{B} + \overline{C} + A)}; \text{ second application of DeMorgan's} \\
 &\quad \text{theorem and product-of-sums form}
 \end{aligned}$$

B.9 No solution provided.

B.10

- $x_0\overline{x_1}\overline{x_2} + \overline{x_0}x_1\overline{x_2} + \overline{x_0}\overline{x_1}x_2$
- $\overline{x_0}\overline{x_1}x_2 + x_0x_1\overline{x_2} + x_0\overline{x_1}x_2 + \overline{x_0}x_1x_2$
- $\overline{x_2}(x_1 + \overline{x_0})$
- $x_2(\overline{x_1} + \overline{x_0})$

B.11 No solution provided.

B.12

- $\overline{x_2}y_2 + x_2y_2\overline{x_1}y_1 + x_2y_2x_1y_1\overline{x_0}y_0$
- $x_2y_2 + x_2y_2\overline{x_1}y_1 + x_2y_2x_1y_1\overline{x_0}y_0$
- $(x_2y_2 + \overline{x_2}y_2)(x_1y_1 + \overline{x_1}y_1)(x_0y_0 + \overline{x_0}y_0)$

B.13

Inputs			Output
A	B	S	C
0	X	0	0
1	X	0	1
X	0	1	0
X	1	1	1

B.14 Two multiplexors, control is S , output of one is C ($S = 0$ selects B , $S = 1$ selects A) and the other is D ($S = 0$ selects A , $S = 1$ selects B).

B.15–17

A	B	C	D	Output
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
1	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

B.18 The first is the flip-flop (it changes only when the clock just went from low to high) and the second is a latch.

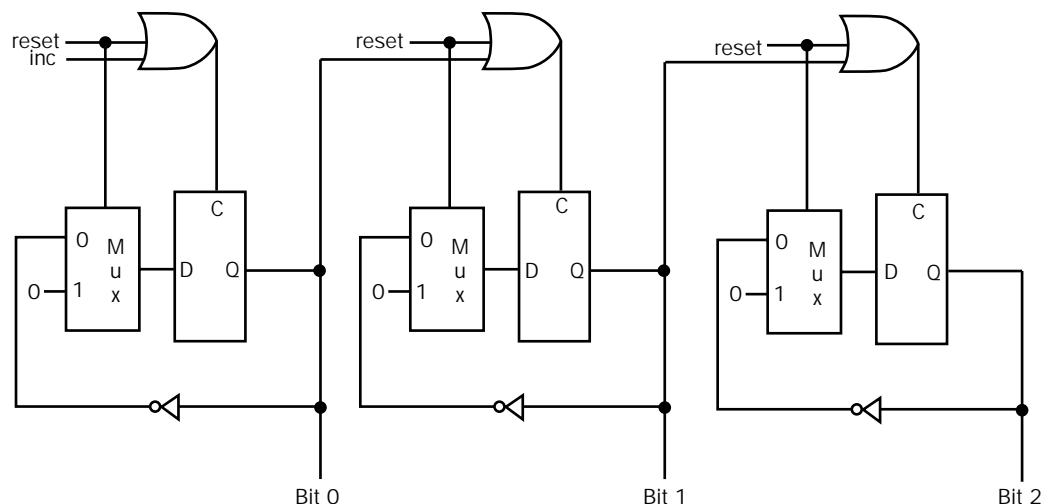
B.19 There will be no difference as long as *D* never changes when *C* is asserted (with respect to the D latch).

B.20 The key is that there are two multiplexors needed and four D flip-flops (should provide diagram).

B.21 The point here is that there are two states with the same output (the middle light on) because the next light is either the left or the right one. This example demonstrates that outputs and states are different.

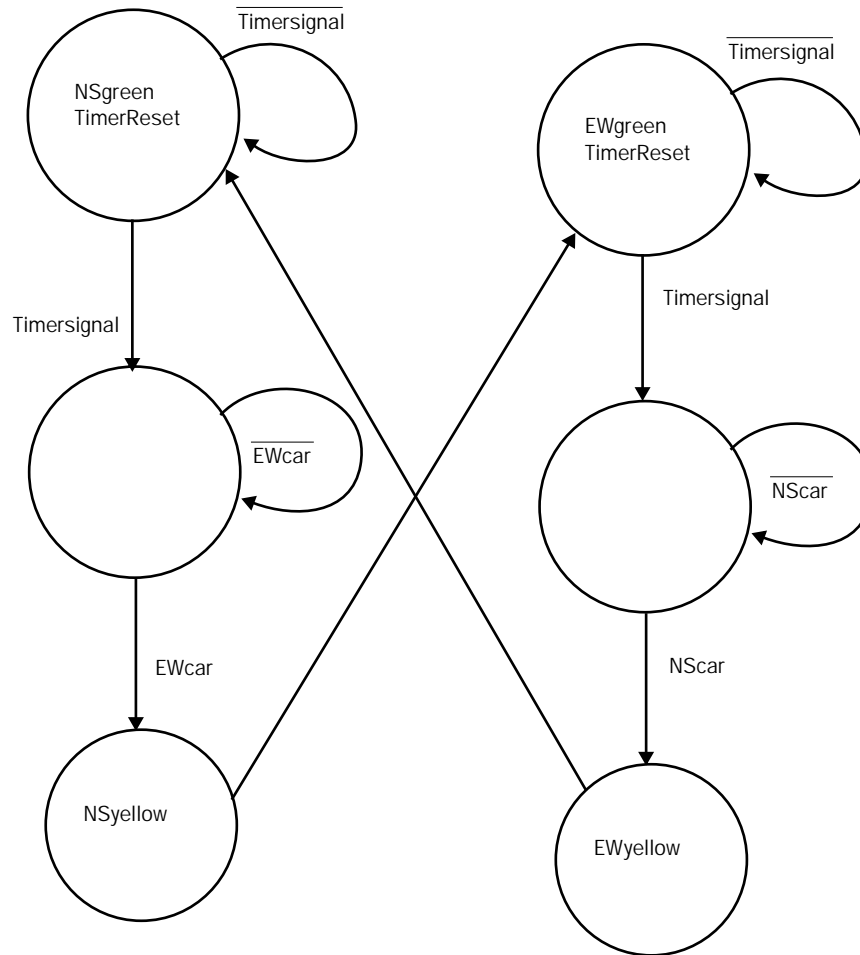
B.22 No solution provided.

B.23 The basic idea is to build a single bit that toggles first. Then build the counter by cascading these 1-bit counters using the output of each stage to cause the toggle of the next; we assume that both *inc* and *reset* are not high at the same time. This circuit assumes falling-edge triggered D flip-flops.



B.24 No solution provided.

B.25 We assume that the light changes only if a car has arrived from the opposite direction and the 30-second timer has expired. Here is the FSM. We make a small simplification and actually make the light cycle in 34-second increments. This could be improved by detecting a car and the timer expiration at the same time, but this still means that when a car arrives it may still take up to 4 seconds before the yellow light is signaled.



B.26 No solution provided.

B.27 No solution provided.

B.28 No solution provided.

C

Solutions

C.1 No solution provided.

C.2 See the solution to Exercise 5.6 and see Figure C.5, which shows a PLA implementing much of the control function logic for a single-cycle MIPS. The instruction `jal` changes the PC and stores PC+4 in register `$ra` (register 31). The Jump control signal manages the correct change to PC. The two-input multiplexors controlled by `RegDst` and `MemToReg` can be expanded to include `$ra` and PC+4, respectively, as inputs. Because `jal` does not use the ALU, all ALU-related controls (`ALUSrc` and `ALUOp`) can be don't cares, so no product terms are needed. The existing Branch, MemRead, and MemWrite controls must be properly set.

Looking at Figure C.5, the R-format, `lw`, `sw`, and `beq` product terms are needed to generate the control signals:

`RegDst` (now must be 2 bits to control the mux with inputs `Rt`, `Rd`, and new input `Sra` (31) in Figure 5.29)

`MemToReg` (now must be 2 bits to control the mux with inputs Read data, ALU result, and new input PC+4 in Figure 5.29)

`MemRead` (= 0)

`MemWrite` (= 0)

`Branch` (= 0)

To generate the control signal `Jump` (controls the upper right multiplexor in Figure 5.29), a new minterm is needed. This new minterm will also be sufficient to generate the additional bit for both the `RegDst` and `MemToReg` control signals.

Thus, five minterms (R-format, `lw`, `sw`, `beq`, and `jump`) are needed.

C.3 See the solution to Exercise 5.5. The control signals `RegDst`, `ALUSrc`, `ALUOp` (2 bits), `MemToReg`, `RegWrite`, `Branch`, `MemRead`, and `MemWrite` must all be set appropriately. From Figure C.5 all four product terms R-format, `lw`, `sw`, and `beq` are used to generate these nine control signals.

C.4 No solution provided.

C.5 No solution provided.

C.6 No solution provided.

C.7 No solution provided.