

**Computer  
Organization and  
Embedded Systems  
5th Edition  
Carl Hamacher**

**Solutions Manual**

# Chapter 1

## Basic Structure of Computers

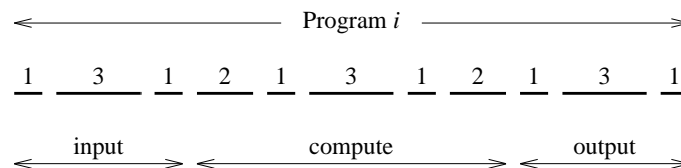
- 1.1.
- Transfer the contents of register PC to register MAR
  - Issue a Read command to memory, and then wait until it has transferred the requested word into register MDR
  - Transfer the instruction from MDR into IR and decode it
  - Transfer the address LOCA from IR to MAR
  - Issue a Read command and wait until MDR is loaded
  - Transfer contents of MDR to the ALU
  - Transfer contents of R0 to the ALU
  - Perform addition of the two operands in the ALU and transfer result into R0
  - Transfer contents of PC to ALU
  - Add 1 to operand in ALU and transfer incremented address to PC
- 1.2.
- First three steps are the same as in Problem 1.1
  - Transfer contents of R1 and R2 to the ALU
  - Perform addition of two operands in the ALU and transfer answer into R3
  - Last two steps are the same as in Problem 1.1
- 1.3. (a)

Load A,R0  
 Load B,R1  
 Add R0,R1  
 Store R1,C

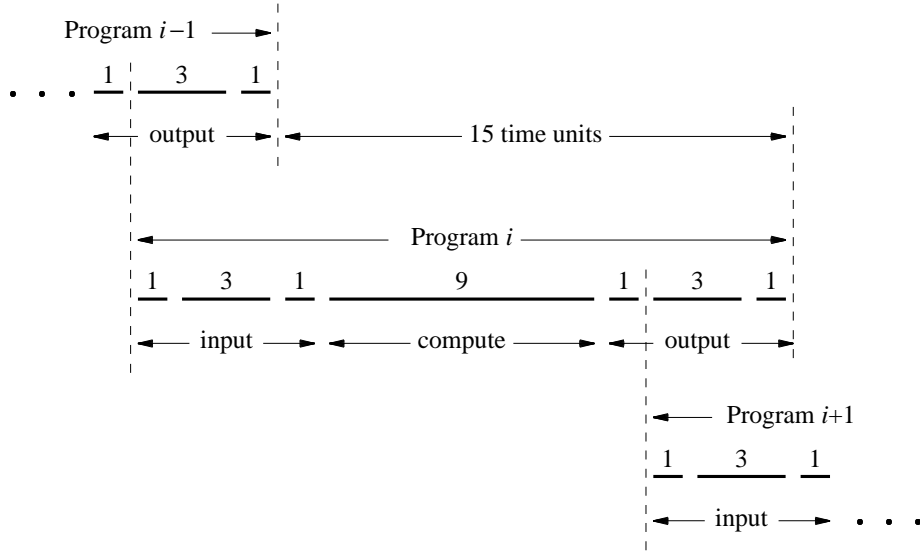
(b) Yes;

Move B,C  
 Add A,C

- 1.4. (a) Non-overlapped time for Program  $i$  is 19 time units composed as:



Overlapped time is composed as:



Time between successive program completions in the overlapped case is 15 time units, while in the non-overlapped case it is 19 time units.

Therefore, the ratio is 15/19.

(b) In the discussion in Section 1.5, the overlap was only between input and output of two successive tasks. If it is possible to do output from job  $i-1$ , compute for job  $i$ , and input to job  $i+1$  at the same time, involving all three units of printer, processor, and disk continuously, then potentially the ratio could be reduced toward 1/3. The OS routines needed to coordinate multiple unit activity cannot be fully overlapped with other activity because they use the processor. Therefore, the ratio cannot actually be reduced to 1/3.

- 1.5. (a) Let  $T_R = (N_R \times S_R) / R_R$  and  $T_C = (N_C \times S_C) / R_C$  be execution times on the RISC and CISC processors, respectively. Equating execution times and clock rates, we have

$$1.2 N_R = 1.5 N_C$$

Then

$$N_C / N_R = 1.2 / 1.5 = 0.8$$

Therefore, the largest allowable value for  $N_C$  is 80% of  $N_R$ .

(b) In this case

$$1.2 N_R / 1.15 = 1.5 N_C / 1.00$$

Then

$$N_C / N_R = 1.2 / (1.15 \times 1.5) = 0.696$$

Therefore, the largest allowable value for  $N_C$  is 69.6% of  $N_R$ .

- 1.6. (a) Let cache access time be 1 and main memory access time be 20. Every instruction that is executed must be fetched from the cache, and an additional fetch from the main memory must be performed for 4% of these cache accesses. Therefore,

$$\text{Speedup factor} = \frac{1.0 \times 20}{(1.0 \times 1) + (0.04 \times 20)} = 11.1$$

(b)

$$\text{Speedup factor} = \frac{1.0 \times 20}{(1.0 \times 1) + (0.02 \times 20)} = 16.7$$

# Chapter 2

## Machine Instructions and Programs

2.1. The three binary representations are given as:

Decimal values	Sign-and-magnitude representation	1's-complement representation	2's-complement representation
5	0000101	0000101	0000101
-2	1000010	1111101	1111110
14	0001110	0001110	0001110
-10	1001010	1110101	1110110
26	0011010	0011010	0011010
-19	1010011	1101100	1101101
51	0110011	0110011	0110011
-43	1101011	1010100	1010101

2.2. (a)

(a)	00101 + 01010 ----- 01111 no overflow	(b)	00111 + 01101 ----- 10100 overflow	(c)	10010 + 01011 ----- 11101 no overflow
(d)	11011 + 00111 ----- 00010 no overflow	(e)	11101 + 11000 ----- 10101 no overflow	(f)	10110 + 10011 ----- 01001 overflow

(b) To subtract the second number, form its 2's-complement and add it to the first number.

(a)	00101 + 10110 ----- 11011 no overflow	(b)	00111 + 10011 ----- 11010 no overflow	(c)	10010 + 10101 ----- 00111 overflow
(d)	11011 + 11001 ----- 10100 no overflow	(e)	11101 + 01000 ----- 00101 no overflow	(f)	10110 + 01101 ----- 00011 no overflow

- 2.3. No; any binary pattern can be interpreted as a number or as an instruction.
- 2.4. The number 44 and the ASCII punctuation character “comma”.
- 2.5. Byte contents in hex, starting at location 1000, will be 4A, 6F, 68, 6E, 73, 6F, 6E. The two words at 1000 and 1004 will be 4A6F686E and 736F6E6E. Byte 1007 (shown as XX) is unchanged. (See Section 2.6.3 for hex notation.)
- 2.6. Byte contents in hex, starting at location 1000, will be 4A, 6F, 68, 6E, 73, 6F, 6E. The two words at 1000 and 1004 will be 6E686F4A and XX6E6F73. Byte 1007 (shown as XX) is unchanged. (See section 2.6.3 for hex notation.)
- 2.7. Clear the high-order 4 bits of each byte to 0000.
- 2.8. A program for the expression is:

Load	A
Multiply	B
Store	RESULT
Load	C
Multiply	D
Add	RESULT
Store	RESULT

2.9. Memory word location J contains the number of tests,  $j$ , and memory word location N contains the number of students,  $n$ . The list of student marks begins at memory word location LIST in the format shown in Figure 2.14. The parameter  $\text{Stride} = 4(j + 1)$  is the distance in bytes between scores on a particular test for adjacent students in the list.

The Base with index addressing mode (R1,R2) is used to access the scores on a particular test. Register R1 points to the test score for student 1, and R2 is incremented by Stride in the inner loop to access scores on the same test by successive students in the list.

	Move	J,R4	Compute and place $\text{Stride} = 4(j + 1)$
	Increment	R4	into register R4.
	Multiply	#4,R4	
	Move	#LIST,R1	Initialize base register R1 to the
	Add	#4,R1	location of the test 1 score for student 1.
	Move	#SUM,R3	Initialize register R3 to the location
			of the sum for test 1.
OUTER	Move	J,R10	Initialize outer loop counter R10 to $j$ .
	Move	N,R11	Initialize inner loop counter R11 to $n$ .
	Clear	R2	Clear index register R2 to zero.
INNER	Clear	R0	Clear sum register R0 to zero.
	Add	(R1,R2),R0	Accumulate the sum of test scores in R0.
	Add	R4,R2	Increment index register R2 by Stride value.
	Decrement	R11	Check if all student scores on current
	Branch>0	INNER	test have been accumulated.
	Move	R0,(R3)	Store sum of current test scores and
	Add	#4,R3	increment sum location pointer.
	Add	#4,R1	Increment base register to next test
			score for student 1.
	Decrement	R10	Check if the sums for all tests have
	Branch>0	OUTER	been computed.

2.10. (a)

			Memory accesses
			<hr/>
	Move	#AVEC,R1	1
	Move	#BVEC,R2	1
	Load	N,R3	2
	Clear	R0	1
LOOP	Load	(R1)+,R4	2
	Load	(R2)+,R5	2
	Multiply	R4,R5	1
	Add	R5,R0	1
	Decrement	R3	1
	Branch>0	LOOP	1
	Store	R0,DOTPROD	2

(b)  $k_1 = 1 + 1 + 2 + 1 + 2 = 7$ ; and  $k_2 = 2 + 2 + 1 + 1 + 1 + 1 = 8$

2.11. (a) The original program in Figure 2.33 is efficient on this task.

(b)  $k_1 = 7$ ; and  $k_2 = 7$

This is only better than the program in Problem 2.10(a) by a small amount.

2.12. The dot product program in Figure 2.33 uses five registers. Instead of using R0 to accumulate the sum, the sum can be accumulated directly into DOTPROD. This means that the last Move instruction in the program can be removed, but DOTPROD is read and written on each pass through the loop, significantly increasing memory accesses. The four registers R1, R2, R3, and R4, are still needed to make this program efficient, and they are all used in the loop. Suppose that R1 and R2 are retained as pointers to the A and B vectors. Counter register R3 and temporary storage register R4 could be replaced by memory locations in a 2-register machine; but the number of memory accesses would increase significantly.

2.13. 1220, part of the instruction, 5830, 4599, 1200.



2.14. Linked list version of the student test scores program:

	Move	#1000,R0
	Clear	R1
	Clear	R2
	Clear	R3
LOOP	Add	8(R0),R1
	Add	12(R0),R2
	Add	16(R0),R3
	Move	4(R0),R0
	Branch>0	LOOP
	Move	R1,SUM1
	Move	R2,SUM2
	Move	R3,SUM3

2.15. Assume that the subroutine can change the contents of any register used to pass parameters.

### Subroutine

	Move	R5,−(SP)	Save R5 on stack.
	Multiply	#4,R4	Use R4 to contain distance in bytes (Stride) between successive elements in a column.
	Multiply	#4,R1	Byte distances from A(0,0) to A(0,x) and A(0,y) placed in R1 and R2.
	Multiply	#4,R2	
LOOP	Move	(R0,R1),R5	Add corresponding column elements.
	Add	R5,(R0,R2)	
	Add	R4,R1	Increment column element pointers by Stride value.
	Add	R4,R2	
	Decrement	R3	Repeat until all elements are added.
	Branch>0	LOOP	
	Move	(SP)+,R5	Restore R5.
	Return		Return to calling program.

- 2.16. The assembler directives `ORIGIN` and `DATAWORD` cause the object program memory image constructed by the assembler to indicate that 300 is to be placed at memory word location 1000 at the time the program is loaded into memory prior to execution.

The `Move` instruction places 300 into memory word location 1000 when the instruction is executed as part of a program.

- 2.17. (a)

```
Move  (R5)+,R0
Add   (R5)+,R0
Move  R0,-(R5)
```

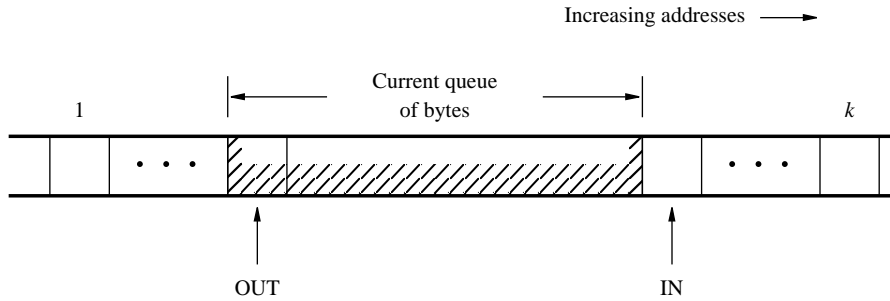
- (b)

```
Move  16(R5),R3
```

- (c)

```
Add  #40,R5
```

- 2.18. (a) Wraparound must be used. That is, the next item must be entered at the beginning of the memory region, assuming that location is empty.
- (b) A current queue of bytes is shown in the memory region from byte location 1 to byte location  $k$  in the following diagram.



The IN pointer points to the location where the next byte will be appended to the queue. If the queue is not full with  $k$  bytes, this location is empty, as shown in the diagram.

The OUT pointer points to the location containing the next byte to be removed from the queue. If the queue is not empty, this location contains a valid byte, as shown in the diagram.

Initially, the queue is empty and both IN and OUT point to location 1.

(c) Initially, as stated in Part b, when the queue is empty, both the IN and OUT pointers point to location 1. When the queue has been filled with  $k$  bytes and none of them have been removed, the OUT pointer still points to location 1. But the IN pointer must also be pointing to location 1, because (following the wraparound rule) it must point to the location where the next byte will be appended. Thus, in both cases, both pointers point to location 1; but in one case the queue is empty, and in the other case it is full.

(d) One way to resolve the problem in Part (c) is to maintain at least one empty location at all times. That is, an item cannot be appended to the queue if  $([IN] + 1) \text{ Modulo } k = [OUT]$ . If this is done, the queue is empty only when  $[IN] = [OUT]$ .

(e) Append operation:

- $LOC \leftarrow [IN]$
- $IN \leftarrow ([IN] + 1) \text{ Modulo } k$
- If  $[IN] = [OUT]$ , queue is full. Restore contents of IN to contents of LOC and indicate failed append operation, that is, indicate that the queue was full. Otherwise, store new item at LOC.

Remove operation:

- If  $[IN] = [OUT]$ , the queue is empty. Indicate failed remove operation, that is, indicate that the queue was empty. Otherwise, read the item pointed to by  $OUT$  and perform  $OUT \leftarrow ([OUT] + 1) \text{ Modulo } k$ .

2.19. Use the following register assignment:

R0 – Item to be appended to or removed from queue  
R1 – IN pointer  
R2 – OUT pointer  
R3 – Address of beginning of queue area in memory  
R4 – Address of end of queue area in memory  
R5 – Temporary storage for  $[IN]$  during append operation

Assume that the queue is initially empty, with  $[R1] = [R2] = [R3]$ .

The following APPEND and REMOVE routines implement the procedures required in Part (e) of Problem 2.18.

APPEND routine:

	Move	R1,R5	
	Increment	R1	Increment IN pointer
	Compare	R1,R4	Modulo $k$ .
	Branch $\geq 0$	CHECK	
	Move	R3,R1	
CHECK	Compare	R1,R2	Check if queue is full.
	Branch=0	FULL	
	MoveByte	R0,(R5)	If queue not full, append item.
	Branch	CONTINUE	
FULL	Move	R5,R1	Restore IN pointer and send
	Call	QUEUEFULL	message that queue is full.
CONTINUE	...		

REMOVE routine:

	Compare	R1,R2	Check if queue is empty.
	Branch=0	EMPTY	If empty, send message.
	MoveByte	(R2)+,R0	Otherwise, remove byte and
	Compare	R2,R4	increment R2 Modulo $k$ .
	Branch $\geq 0$	CONTINUE	
	Move	R3,R2	
	Branch	CONTINUE	
EMPTY	Call	QUEUEEMPTY	
CONTINUE	...		

- 2.20. (a) Neither nesting nor recursion are supported.  
 (b) Nesting is supported, because different Call instructions will save the return address at different memory locations. Recursion is not supported.  
 (c) Both nesting and recursion are supported.
- 2.21. To allow nesting, the first action performed by the subroutine is to save the contents of the link register on a stack. The Return instruction pops this value into the program counter. This supports recursion, that is, when the subroutine calls itself.
- 2.22. Assume that register SP is used as the stack pointer and that the stack grows toward lower addresses. Also assume that the memory is byte-addressable and that all stack entries are 4-byte words. Initially, the stack is empty. Therefore, SP contains the address [LOWERLIMIT] + 4. The routines CALLSUB and RETRN must check for the stack full and stack empty cases as shown in Parts (b) and (a) of Figure 2.23, respectively.

CALLSUB	Compare	UPPERLIMIT,SP
	Branch $\leq 0$	FULLERROR
	Move	RL, -(SP)
	Branch	(R1)

RETRN	Compare	LOWERLIMIT,SP
	Branch $> 0$	EMPTYERROR
	Move	(SP)+,PC

- 2.23. If the ID of the new record matches the ID of the Head record of the current list, the new record will be inserted as the new Head. If the ID of the new record matches the ID of a later record in the current list, the new record will be inserted immediately after that record, including the case where the matching record is the Tail record. In this latter case, the new record becomes the new Tail record.

Modify Figure 2.37 as follows:

- Add the following instruction as the first instruction of the subroutine:

INSERTION	Move	#0, ERROR	Anticipate successful insertion of the new record.
	Compare	#0, RHEAD	(Existing instruction.)

- After the second Compare instruction, insert the following three instructions:

	Branch $\neq$ 0	CONTINUE1	Three new instructions.
	Move	RHEAD, ERROR	
	Return		
CONTINUE1	Branch $>$ 0	SEARCH	(Existing instruction.)

- After the fourth Compare instruction, insert the following three instructions:

	Branch $\neq$ 0	CONTINUE2	Three new instructions.
	Move	RNEXT, ERROR	
	Return		
CONTINUE2	Branch $<$ 0	INSERT	(Existing instruction.)

- 2.24. If the list is empty, the result is unpredictable because the first instruction will compare the ID of the new record to the contents of memory location zero. If the list is not empty, the following happens. If the contents of RIDNUM are less than the ID number of the Head record, the Head record will be deleted. Otherwise, the routine loops until register RCURRENT points to the Tail record. Then RNEXT gets loaded with zero by the instruction at LOOP, and the result is unpredictable.

Replace Figure 2.38 with the following code:

DELETION	Compare #0, RHEAD	If the list is empty,
	Branch $\neq$ 0 CHECKHEAD	return with RIDNUM unchanged.
	Return	
CHECKHEAD	Compare (RHEAD), RIDNUM	Check if Head record
	Branch $\neq$ 0 CONTINUE1	is to be deleted and
	Move 4(RHEAD), RHEAD	perform deletion if it
	Move #0, RIDNUM	is, returning with zero
	Return	in RIDNUM.
CONTINUE1	Move RHEAD, RCURRENT	Otherwise, continue searching.
LOOP	Move 4(CURRENT), RNEXT	
	Compare #0, RNEXT	If all records checked,
	Branch $\neq$ 0 CHECKNEXT	return with IDNUM unchanged.
	Return	
CHECKNEXT	Compare (RNEXT), RIDNUM	Check if next record is
	Branch $\neq$ 0 CONTINUE2	to be deleted and perform
	Move 4(RNEXT), RTEMP	deletion if it is,
	Move RTEMP, 4(RCURRENT)	returning with zero
	Move #0, RIDNUM	in RIDNUM.
	Return	
CONTINUE2	Move RNEXT, RCURRENT	Otherwise, continue
	Branch LOOP	the search.

## Chapter 3

# ARM, Motorola, and Intel Instruction Sets

### PART I: ARM

- 3.1. (a) R8, R9, and R10, contain 1, 2, and 3, respectively.
- (b) The values 20 and 30 are pushed onto a stack pointed to by R1 by the two Store instructions, and they occupy memory locations 1996 and 1992, respectively. They are then popped off the stack into R8 and R9. Finally, the Subtract instruction results in 10 ( $30 - 20$ ) being stored in R10. The stack pointer R1 is returned to its original value of 2000.
- (c) The numbers in memory locations 1016 and 1020 are loaded into R4 and R5, respectively. These two numbers are then added and the sum is placed in register R4. The final address value in R2 is 1024.
- 3.2. (b) A memory operand cannot be referenced in a Subtract instruction.
- (d) The immediate value 257 is 100000001 in binary, and is thus too long to fit in the 8-bit immediate field. Note that it cannot be generated by the rotation of any 8-bit value.
- 3.3. The following two instructions perform the desired operation:
- ```
MOV    R0,R0,LSL #24
MOV    R0,R0,ASR #24
```
- 3.4. Use register R0 as a counter register and R1 as a work register.

|      |      |              |                                                                                              |
|------|------|--------------|----------------------------------------------------------------------------------------------|
|      | MOV  | R0,#32       | Load R0 with count value 32.                                                                 |
|      | MOV  | R1,#0        | Clear register R1 to zero.                                                                   |
| LOOP | MOV  | R2,R2,LSL #1 | Shift contents of R2 left<br>one bit position, moving the<br>high-order bit into the C flag. |
|      | MOV  | R1,R1,RRX    | Rotate R1 right one bit<br>position, including the C flag,<br>as shown in Figure 2.32d.      |
|      | SUBS | R0,R0,#1     | Check if finished.                                                                           |
|      | BGT  | LOOP         |                                                                                              |
|      | MOV  | R2,R1        | Load reversed pattern<br>back into R2.                                                       |

3.5. Program trace:

| TIME                       | R0  | R1 | R2        |
|----------------------------|-----|----|-----------|
| after 1st execution of BGT | 3   | 4  | NUM1 + 4  |
| after 2nd execution of BGT | -14 | 3  | NUM1 + 8  |
| after 3rd execution of BGT | 13  | 2  | NUM1 + 12 |

3.6. Assume bytes are unsigned 8-bit values.

|      |        |            |                                |
|------|--------|------------|--------------------------------|
|      | LDR    | R0,N       | R0 is list counter             |
|      | ADR    | R1,X       | R1 points to X list            |
|      | ADR    | R2,Y       | R2 points to Y list            |
|      | ADR    | R3,LARGER  | R3 points to LARGER list       |
| LOOP | LDRB   | R4,[R1],#1 | Load X list byte into R4       |
|      | LDRB   | R5,[R2],#1 | Load Y list byte into R5       |
|      | CMP    | R4,R5      | Compare bytes                  |
|      | STRHSB | R4,[R3],#1 | Store X byte if larger or same |
|      | STRLOB | R5,[R3],#1 | Store Y byte if larger         |
|      | SUBS   | R0,R0,#1   | Check if finished              |
|      | BGT    | LOOP       |                                |

3.7. The inner loop checks for a match at each possible position.

|         |      |              |                                |
|---------|------|--------------|--------------------------------|
|         | LDR  | R0,N         | Compute outer loop count       |
|         | LDR  | R1,M         | and store in R2.               |
|         | SUB  | R2,R0,R1     |                                |
|         | ADD  | R2,R2,#1     |                                |
|         | ADR  | R3,STRING    | Use R3 and R4 as base          |
|         | ADR  | R4,SUBSTRING | pointers for each match.       |
| OUTER   | MOV  | R5,R3        | Use R5 and R6 as running       |
|         | MOV  | R6,R4        | pointers for each match.       |
|         | LDR  | R7,M         | Initialize inner loop counter. |
| INNER   | LDRB | R0,[R5],#1   | Compare bytes.                 |
|         | LDRB | R1,[R6],#1   |                                |
|         | CMP  | R0,R1        |                                |
|         | BNE  | NOMATCH      | If not equal, go next.         |
|         | SUBS | R7,R7,#1     | Check if all bytes compared.   |
|         | BGT  | INNER        |                                |
|         | MOV  | R0,R3        | If substring matches, load     |
|         | B    | NEXT         | its position into R0 and exit. |
| NOMATCH | ADD  | R3,R3,#1     | Go to next substring.          |
|         | SUBS | R2,R2,#1     | Check if all positions tried.  |
|         | BGT  | OUTER        |                                |
|         | MOV  | R0,#0        | If yes, load zero into         |
| NEXT    | ...  |              | R0 and exit.                   |



- 3.8. This solution assumes that the last number in the series of  $n$  numbers can be represented in a 32-bit word, and that  $n > 2$ .

|      |      |            |                              |
|------|------|------------|------------------------------|
|      | MOV  | R0,N       | Use R0 to count numbers      |
|      | SUB  | R0,R0,#2   | generated after 1.           |
|      | ADR  | R1,MEMLOC  | Use R1 as memory pointer.    |
|      | MOV  | R2,#0      | Store first two numbers,     |
|      | STR  | R2,[R1],#4 | 0 and 1, from R2             |
|      | MOV  | R3,#1      | and R3 into memory.          |
|      | STR  | R3,[R1],#4 |                              |
| LOOP | ADD  | R3,R2,R3   | Starting with number $i - 1$ |
|      | STR  | R3,[R1],#4 | in R2 and $i$ in R3, compute |
|      |      |            | and place $i + 1$ in R3      |
|      |      |            | and store in memory.         |
|      | SUB  | R2,R3,R2   | Recover old $i$ and place    |
|      |      |            | in R2.                       |
|      | SUBS | R0,R0,#1   | Check if all numbers         |
|      | BGT  | LOOP       | have been computed.          |

- 3.9. Let R0 point to the ASCII word beginning at location WORD. To change to uppercase, we need to change bit  $b_5$  from 1 to 0.

|      |        |            |                           |
|------|--------|------------|---------------------------|
| NEXT | LDRB   | R1,[R0]    | Get character.            |
|      | CMP    | #&20,R1    | Check if space character. |
|      | ANDNE  | #&DF,R1    | If not space: clear       |
|      | STRNEB | R1,[R0],#1 | bit 5, store              |
|      | BNE    | NEXT       | converted character,      |
|      |        |            | get next character.       |

- 3.10. Memory word location J contains the number of tests,  $j$ , and memory word location N contains the number of students,  $n$ . The list of student marks begins at memory word location LIST in the format shown in Figure 2.14. The parameter  $\text{Stride} = 4(j + 1)$  is the distance in bytes between scores on a particular test for adjacent students in the list.

The Post-indexed addressing mode  $[\text{R2}], \text{R3}, \text{LSL} \#2$  is used to access the successive scores on a particular test in the inner loop. The value in register R2 before each entry to the inner loop is the address of the score on a particular test for the first student. Register R3 contains the value  $j + 1$ . Therefore, register R2 is incremented by the Stride parameter on each pass through the inner loop.

|       |      |                   |                               |
|-------|------|-------------------|-------------------------------|
|       | LDR  | R3,J              | Load $j + 1$ into R3 to       |
|       | ADD  | R3,R3,#1          | be used as an address offset. |
|       | ADR  | R4,SUM            | Initialize R4 to the sum      |
|       |      |                   | location for test 1.          |
|       | ADR  | R5,LIST           | Load address of test 1 score  |
|       | ADD  | R5,R5,#4          | for student 1 into R5.        |
|       | LDR  | R6,J              | Initialize outer loop counter |
|       |      |                   | R6 to $j$ .                   |
| OUTER | LDR  | R7,N              | Initialize inner loop         |
|       |      |                   | counter R7 to $n$ .           |
|       | MOV  | R2,R5             | Initialize base register R2   |
|       |      |                   | to location of student 1 test |
|       |      |                   | score for next inner loop     |
|       |      |                   | sum computation.              |
|       | MOV  | R0,#0             | Clear sum accumulator         |
|       |      |                   | register R0.                  |
| INNER | LDR  | R1,[R2],R3,LSL #2 | Load test score into R1       |
|       |      |                   | and increment R2 by Stride to |
|       |      |                   | point to next test score.     |
|       | ADD  | R0,R0,R1          | Accumulate score into R0.     |
|       | SUBS | R7,R7,#1          | Check if all student scores   |
|       | BGT  | INNER             | for current test are added.   |
|       | STR  | R0,[R4],#4        | Store sum in memory.          |
|       | ADD  | R5,R5,#4          | Increment R5 to next test     |
|       |      |                   | score for student 1.          |
|       | SUBS | R6,R6,#1          | Check if sums for all test    |
|       | BGT  | OUTER             | scores have been accumulated. |

- 3.11. Assume that the subroutine can change the contents of any registers used to pass parameters.

|      |      |                   |                                                                      |
|------|------|-------------------|----------------------------------------------------------------------|
|      | STR  | R5,[R13,#4]!      | Save [R5] on stack.                                                  |
|      | ADD  | R1,R0,R1,LSL #2   | Load address of A(0,x) into R1.                                      |
|      | ADD  | R2,R0,R2,LSL #2   | Load address of A(0,y) into R2.                                      |
| LOOP | LDR  | R5,[R1],R4,LSL #2 | Load [A(i,x)] into R5<br>and increment pointer R1<br>by Stride = 4m. |
|      | LDR  | R0,[R2]           | Load [A(i,y)] into R0.                                               |
|      | ADD  | R0,R0,R5          | Add corresponding column entries.                                    |
|      | STR  | R0,[R2],R4,LSL #2 | Store sum in A(i,y) and<br>increment pointer R2 by Stride.           |
|      | SUBS | R3,R3,#1          | Repeat loop until all                                                |
|      | BGT  | LOOP              | entries have been added.                                             |
|      | LDR  | R5,[R13],#4       | Restore [R5] from stack.                                             |
|      | MOV  | R15,R14           | Return.                                                              |

- 3.12. This program is similar to Figure 3.9, and makes the same assumptions about register usage and status word bit locations.

|      |      |              |                                                           |
|------|------|--------------|-----------------------------------------------------------|
|      | LDR  | R0,N         | Use R0 as the loop counter<br>for reading $n$ characters. |
| READ | LDR  | R3,[R1]      | Load [INSTATUS] and                                       |
|      | TST  | R3,#8        | wait for character.                                       |
|      | BEQ  | READ         |                                                           |
|      | LDRB | R3,[R1,#4]   | Read character and push                                   |
|      | STRB | R3,[R6,#-1]! | onto stack.                                               |
| ECHO | LDR  | R4,[R2]      | Load [OUTSTATUS] and                                      |
|      | TST  | R4,#8        | wait for display.                                         |
|      | BEQ  | ECHO         |                                                           |
|      | STRB | R3,[R2,#4]   | Send character<br>to display.                             |
|      | SUBS | R0,R0,#1     | Repeat until $n$                                          |
|      | BGT  | READ         | characters read.                                          |

- 3.13. Assume that most of the time between successive characters being struck is spent in the three-instruction wait loop that starts at location READ. The BEQ READ instruction is executed once every 60 ns while this loop is being executed. There are  $10^9/10 = 10^8$  ns between successive characters. Therefore, the BEQ READ instruction is executed  $10^8/60 = 1.6666 \times 10^6$  times per character entered.

### 3.14. Main Program

|          |      |            |                                    |
|----------|------|------------|------------------------------------|
| READLINE | BL   | GETCHAR    | Call character read subroutine.    |
|          | STRB | R3,[R0],#1 | Store character in memory.         |
|          | BL   | PUTCHAR    | Call character display subroutine. |
|          | TEQ  | R3,#CR     | Check for end-of-line character.   |
|          | BNE  | READLINE   |                                    |

#### Subroutine GETCHAR

|         |      |            |                         |
|---------|------|------------|-------------------------|
| GETCHAR | LDR  | R3,[R1]    | Wait for character.     |
|         | TST  | R3,#8      |                         |
|         | BEQ  | GETCHAR    |                         |
|         | LDRB | R3,[R1,#4] | Load character into R3. |
|         | MOV  | R15,R14    | Return.                 |

#### Subroutine PUTCHAR

|         |       |               |                            |
|---------|-------|---------------|----------------------------|
| PUTCHAR | STMFD | R13!,{R4,R14} | Save R4 and Link register. |
| DISPLAY | LDR   | R4,[R2]       | Wait for display.          |
|         | TST   | R4,#8         |                            |
|         | BEQ   | DISPLAY       |                            |
|         | STRB  | R3,[R2,#4]    | Send character to display. |
|         | LDMFD | R13!,{R4,R15} | Restore R4 and Return.     |

- 3.15. Address INSTATUS is passed to GETCHAR on the stack; the character read is passed back in the same stack position. The character to be displayed and the address OUTSTATUS are passed to PUTCHAR on the stack in that order. The stack frame structure shown in Figure 3.13 is used.

#### Main Program

|          |      |              |                                                    |
|----------|------|--------------|----------------------------------------------------|
| READLINE | LDR  | R1,POINTER1  | Load address INSTATUS                              |
|          | STR  | R1,[SP,#-4]! | contained in POINTER1 into R1 and push onto stack. |
|          | BL   | GETCHAR      | Call character read subroutine.                    |
|          | LDRB | R1,[SP]      | Load character from top of                         |
|          | STRB | R1,[R0],#1   | stack and store in memory.                         |
|          | LDR  | R2,POINTER2  | Load address OUTSTATUS                             |
|          | STR  | R2,[SP,#-4]! | contained in POINTER2 into R2 and push onto stack. |
|          | BL   | PUTCHAR      | Call character display subroutine.                 |
|          | ADD  | SP,SP,#8     | Remove parameters from stack.                      |
|          | TEQ  | R1,#CR       | Check for end-of-line character.                   |
|          | BNE  | READLINE     |                                                    |

#### Subroutine GETCHAR

|         |       |                   |                                  |
|---------|-------|-------------------|----------------------------------|
| GETCHAR | STMFD | SP!,{R1,R3,FP,LR} | Save registers.                  |
|         | ADD   | FP,SP,#8          | Load frame pointer.              |
|         | LDR   | R1[FP,#8]         | Load address INSTATUS into R1.   |
| READ    | LDR   | R3,[R1]           | Wait for character.              |
|         | TST   | R3,#8             |                                  |
|         | BEQ   | READ              |                                  |
|         | LDRB  | R3,[R1,#4]        | Load character into R3           |
|         | STRB  | R3,[FP,#8]        | and overwrite INSTATUS on stack. |
|         | LDMFD | SP!,{R1,R3,FP,PC} | Restore registers and Return.    |

#### Subroutine PUTCHAR

|         |       |                   |                               |
|---------|-------|-------------------|-------------------------------|
| PUTCHAR | STMFD | SP!,{R2-R4,FP,LR} | Save registers.               |
|         | ADD   | FP,SP,#12         | Load frame pointer.           |
|         | LDR   | R2,[FP,#8]        | Load address OUTSTATUS into   |
|         | LDR   | R3,[FP,#12]       | R2 and character into R3.     |
| DISPLAY | LDR   | R4,[R2]           | Wait for display.             |
|         | TST   | R4,#8             |                               |
|         | BEQ   | DISPLAY           |                               |
|         | STRB  | R3,[R2,#4]        | Send character to display.    |
|         | LDMFD | SP!,{R2-R4,FP,PC} | Restore registers and Return. |

- 3.16. The first program section reads the characters, stores them in a 3-byte area beginning at CHARSTR, and echoes them to a display. The second section does the conversion to binary and stores the result in BINARY. The I/O device addresses INSTATUS and OUTSTATUS are in registers R1 and R2.

|         |      |                   |                            |
|---------|------|-------------------|----------------------------|
| READ    | ADR  | R0,CHARSTR        | Initialize memory pointer  |
|         | MOV  | R5,#3             | R0 and counter R5.         |
|         | LDR  | R3,[R1]           | Read a character and       |
|         | TST  | R3,#8             | store it in memory.        |
|         | BEQ  | READ              |                            |
| ECHO    | LDRB | R3,[R1,#4]        |                            |
|         | STRB | R3,[R0],#1        |                            |
|         | LDR  | R4,[R2]           | Echo the character         |
|         | TST  | R4,#8             | to the display.            |
|         | BEQ  | ECHO              |                            |
| CONVERT | STRB | R3,[R2,#4]        |                            |
|         | SUBS | R5,R5,#1          | Check if all three         |
|         | BGT  | READ              | characters have been read. |
|         | ADR  | R0,CHARSTR        | Initialize memory pointers |
|         | ADR  | R1,HUNDREDS       | R0, R1, and R2.            |
|         | ADR  | R2,TENS           |                            |
|         | LDRB | R3,[R0],#1        | Load high-order BCD digit  |
|         | AND  | R3,R3,#&F         | into R3.                   |
|         | LDR  | R4,[R1,R3,LSL #2] | Load binary value          |
|         |      |                   | corresponding to decimal   |
|         |      |                   | hundreds value into        |
|         |      |                   | accumulator register R4.   |
|         | LDRB | R3,[R0],#1        | Load middle BCD digit      |
|         | AND  | R3,R3,#&F         | into R3.                   |
|         | LDR  | R3,[R2,R3,LSL #2] | Load binary value          |
|         |      |                   | corresponding to           |
|         |      |                   | decimal tens value         |
|         |      |                   | into register R3.          |
|         | ADD  | R4,R4,R3          | Accumulate into R4.        |
|         | LDRB | R3,[R0],#1        | Load low-order BCD digit   |
|         | AND  | R3,R3,#&F         | into R3.                   |
|         | ADD  | R4,R4,R3          | Accumulate into R4.        |
|         | STR  | R4,BINARY         | Store converted value      |
|         |      |                   | into location BINARY.      |

- 3.17. (a) The names FP, SP, LR, and PC, are used for registers R12, R13, R14, and R15 (frame pointer, stack pointer, link register, and program counter). The 3-byte memory area for the characters begins at address CHARSTR; and the converted binary value is stored at BINARY.

The first subroutine, labeled READCHARS, is patterned after the program in Figure 3.9. It echoes the characters back to a display as well as reading them into memory. The second subroutine is labeled CONVERT.

The stack frame format used is like Figure 3.13.

A possible main program is:

#### Main program

|         |       |               |                             |
|---------|-------|---------------|-----------------------------|
|         | ADR   | R10,CHARSTR   | Load parameters into        |
|         | ADR   | R11,BINARY    | R10 and R11 and             |
|         | STMFD | SP!,{R10,R11} | push onto stack.            |
|         | BL    | READCHARS     | Branch to first subroutine. |
| RTNADDR | ADD   | SP,SP,#8      | Remove two parameters       |
| ...     |       |               | from stack and continue.    |

#### First subroutine READCHARS

|           |       |                   |                          |
|-----------|-------|-------------------|--------------------------|
| READCHARS | STMFD | SP!,{R0–R5,FP,LR} | Save registers on stack. |
|           | ADD   | FP,SP,#28         | Set up frame pointer.    |
|           | LDR   | R0,[FP,#4]        | Load R0, R1, and R2 with |
|           | ADR   | R1,INSTATUS       | parameters.              |
|           | ADR   | R2,OUTSTATUS      |                          |
|           | MOV   | R5,#3             | Same code as             |
|           | ...   |                   | in solution to           |
|           | BGT   | READ              | Problem 3.16.            |
|           | LDR   | R0,[FP,#8]        | Load R0,R1,R2            |
|           | LDR   | R5,[FP,#12]       | and R5 with              |
|           | ADR   | R1,HUNDREDS       | parameters.              |
|           | ADR   | R2,TENS           |                          |
|           | BL    | CONVERT           | Call second subroutine.  |
|           | LDMFD | SP!,{R0–R5,FP,PC} | Return to Main program.  |

## Second subroutine CONVERT

|         |       |                   |                                |
|---------|-------|-------------------|--------------------------------|
| CONVERT | STMFD | SP!,{R3,R4,FP,LR} | Save registers<br>on stack.    |
|         | ADD   | FP,SP,#8          | Set up frame<br>pointer.       |
|         | LDRB  | R3,[R0],#1        | Same code as<br>in solution to |
|         | ...   |                   | Problem 3.16.                  |
|         | ADD   | R4,R4,R3          |                                |
|         | STR   | R4,[R5]           | Store binary<br>number.        |
|         | LDMFD | SP!,{R3,R4,FP,PC} | Return to<br>first subroutine. |

(b) The contents of the top of the stack after the call to the CONVERT routine are:

|      |                |
|------|----------------|
|      |                |
|      | [R0]           |
|      | [R1]           |
|      | [R2]           |
|      | [R3]           |
|      | [R4]           |
|      | [R5]           |
| FP → | [FP]           |
|      | [LR] = RTNADDR |
|      | CHARSTR        |
|      | BINARY         |
|      | Original TOS   |
|      |                |



- 3.18. See the solution to Problem 2.18 for the procedures needed to perform the append and remove operations.

Register assignment:

R0 – Data byte to append to or remove from queue  
R1 – IN pointer  
R2 – OUT pointer  
R3 – Address of first queue byte location  
R4 – Address of last queue byte location ( $= [R3] + k - 1$ )  
R5 – Auxiliary register for address of next appended byte.

Initially, the queue is empty with  $[R1] = [R2] = [R3]$

APPEND routine:

|       |           |                             |
|-------|-----------|-----------------------------|
| MOV   | R5,R1     |                             |
| ADD   | R1,R1,#1  | Increment R1 Modulo $k$ .   |
| CMP   | R1,R4     |                             |
| MOVGT | R1,R3     |                             |
| CMP   | R1,R2     | Check if queue is full.     |
| MOVEQ | R1,R5     | If queue full, restore      |
| BEQ   | QUEUEFULL | IN pointer and send         |
|       |           | message that queue is full. |
| STRB  | R0,[R5]   | If queue not full,          |
|       |           | append byte and continue.   |

REMOVE routine:

|       |            |                          |
|-------|------------|--------------------------|
| CMP   | R1,R2      | Check if queue is empty. |
| BEQ   | QUEUEEMPTY | If empty, send message.  |
| LDRB  | R0,[R2],#1 | Otherwise, remove byte   |
| CMP   | R2,R4      | and increment R2         |
| MOVGT | R2,R3      | Modulo $k$ .             |

- 3.19. Program trace:

| TIME      | R0  | R2   | R3   | LIST | LIST<br>+1 | LIST<br>+2 | LIST<br>+3 | LIST<br>+4 |
|-----------|-----|------|------|------|------------|------------|------------|------------|
| After 1st | 120 | 1004 | 1000 | 106  | 13         | 67         | 45         | 120        |
| After 2nd | 106 | 1003 | 1000 | 67   | 13         | 45         | 106        | 120        |
| After 3rd | 67  | 1002 | 1000 | 45   | 13         | 67         | 106        | 120        |
| After 4th | 45  | 1001 | 1000 | 13   | 45         | 67         | 106        | 120        |

### 3.20. Calling program

```

ADR   R4,LISTN   Pass parameter LISTN to
                  subroutine in R4.
                  Assume LISTN + 4 = LIST.
BL     SORT

```

#### Subroutine SORT

```

SORT   STMFD     R13!,{R0-R3,R5,R14}  Save registers.
        LDR       R0,[R4],#4           Initialize outer loop
        ADD       R2,R4,R0,LSL #2      base register R2
  to LIST + 4n.
        ADD       R5,R4,#4             Load LIST + 4 into
  register R5.
OUTER  LDR       R0,[R2,#-4]!           Comments similar
        MOV       R3,R2                as in Figure 3.15.
INNER  LDR       R1,[R3,#-4]!
        CMP       R1,R0
        STRGT     R1,[R2]
        STRGT     R0,[R3]
        MOVGTE    R0,R1
        CMP       R3,R4
        BNE       INNER
        CMP       R2,R5
        BNE       OUTER
        LDMFD     R13!,{R0-R3,R5,R15}  Restore registers
  and return.

```

3.21. The alternative program from the instruction labeled OUTER to the end is:

|       |       |              |                                                               |
|-------|-------|--------------|---------------------------------------------------------------|
| OUTER | LDRB  | R0,[R2,#-1]! | Load LIST( $j$ ) into R0.                                     |
|       | MOV   | R3,R2        | Initialize inner loop base register<br>R3 to LIST + $n - 1$ . |
|       | MOV   | R6,R2        | Load address of initial largest<br>element into R6.           |
|       | MOV   | R7,R0        | Load initial largest element<br>into R7.                      |
| INNER | LDRB  | R1,[R3,#-1]! | Load LIST( $k$ ) into R1.                                     |
|       | CMP   | R1,R7        | Compare LIST( $k$ ) to current largest.                       |
|       | MOVGT | R6,R3        | Update address and value of<br>largest if LIST( $k$ ) larger. |
|       | MOVGT | R7,R1        |                                                               |
|       | CMP   | R3,R4        | Check if inner loop completed.                                |
|       | BNE   | INNER        |                                                               |
|       | STRB  | R0,[R6]      | Swap; correct code even if no<br>larger element is found.     |
|       | STRB  | R7,[R2]      |                                                               |
|       | CMP   | R2,R5        |                                                               |
|       | BNE   | OUTER        |                                                               |

The advantage of this approach is that the two MOVGT instructions in the inner loop of the alternative program execute faster than the three-instruction interchange code in Figure 3.15*b*.

3.22. The record pointer is register R0, and registers R1, R2, and R3, are used to accumulate the three sums, as in Figure 2.15. Assume that the list is not empty.

|      |     |             |
|------|-----|-------------|
|      | MOV | R0,#1000    |
|      | MOV | R1,#0       |
|      | MOV | R2,#0       |
|      | MOV | R3,#0       |
| LOOP | LDR | R5,[R0,#8]  |
|      | ADD | R1,R1,R5    |
|      | LDR | R5,[R0,#12] |
|      | ADD | R2,R2,R5    |
|      | LDR | R5,[R0,#16] |
|      | ADD | R3,R3,R5    |
|      | LDR | R0,[R0,#4]  |
|      | CMP | R0,#0       |
|      | BNE | LOOP        |
|      | STR | R1,SUM1     |
|      | STR | R2,SUM2     |
|      | STR | R3,SUM3     |

- 3.23. If the ID of the new record matches the ID of the Head record, the new record will become the new Head. If the ID matches that of a later record, it will be inserted immediately after that record, including the case where the matching record is the Tail.

Modify Figure 3.16 as follows:

- Add the following instruction as the first instruction of the subroutine:

```
INSERTION  MOV    R10,#0    Anticipate successful
                                insertion of new record.
```

- After the second CMP instruction, insert the following two instructions:

```
MOVEQ     R10, RHEAD    ID matches that of
MOVEQ     PC, R14       Head record.
```

- After the instruction labeled LOOP, insert the following four instructions:

```
LDR       R0, [RNEXT]
CMP       R0, R1
MOVEQ     R10, RNEXT
MOVEQ     PC, R14
```

- Remove the instruction with the comment “Go further?” because it has already been done in the previous bullet.

- 3.24. If the list is empty, the result is unpredictable because the second instruction compares the new ID with the contents of memory location zero. If the list is not empty, the program continues until RCURRENT points to the Tail record. Then the instruction at LOOP loads zero into RNEXT and the result is unpredictable.

Replace Figure 3.17 with the following code:

|           |       |                      |                                |
|-----------|-------|----------------------|--------------------------------|
| DELETION  | CMP   | RHEAD, #0            | If list is empty, return       |
|           | MOVEQ | PC, R14              | with RIDNUM unchanged.         |
| CHECKHEAD | LDR   | R0, [RHEAD]          | Check if Head record is        |
|           | CMP   | R0, RIDNUM           | to be deleted. If yes,         |
|           | LDREQ | RHEAD, [RHEAD,#4]    | delete it, and then return     |
|           | MOVEQ | RIDNUM, #0           | with zero in RIDNUM.           |
|           | MOVEQ | PC, R14              |                                |
|           | MOV   | RCURRENT, RHEAD      | Otherwise, continue search.    |
| LOOP      | LDR   | RNEXT, [RCURRENT,#4] |                                |
|           | CMP   | RNEXT, #0            | If all records checked, return |
|           | MOVEQ | PC, R14              | with RIDNUM unchanged.         |
|           | LDR   | R0, [RNEXT]          | Is next record the one         |
|           | CMP   | R0, RIDNUM           | to be deleted?                 |
|           | LDREQ | R0, [RNEXT,#4]       | If yes, delete it, and         |
|           | STREQ | R0, [RCURRENT,#4]    | return with zero               |
|           | MOVEQ | RIDNUM, #0           | in RIDNUM.                     |
|           | MOVEQ | PC, R14              |                                |
|           | MOV   | RCURRENT, RNEXT      | Otherwise, loop back and       |
|           | B     | LOOP                 | continue to search.            |

## PART II: 68000

3.25. (a) Location  $\$2000 \leftarrow \$1000 + \$3000 = \$4000$

The instruction occupies two bytes. One memory access is needed to fetch the instruction and 4 to execute it.

(b) Effective Address =  $\$1000 + \$1000 = \$2000$ ,

$D0 \leftarrow \$3000 + \$1000 = \$4000$

4 bytes; 2 accesses to fetch instruction and 2 to execute it.

(c)  $\$2000 \leftarrow \$2000 + \$3000 = \$5000$

6 bytes; 3 accesses to fetch instruction and 4 to execute it.

3.26. (a) `ADDX -(A2),D3`

In Add extended, both the destination and source operands must use the same addressing mode, either register or autodecrement.

(b) `LSR.L #9,D2`

The number of bits shifted must be less than 8.

(c) `MOVE.B 520(A0,D2)`

The offset value requires more than 8 bits. Also, no destination operand is specified.

(d) `SUBA.L 12(A2,PC),A0`

In relative full addressing mode the PC must be specified before the address register.

(e) `CMP.B #254,$12(A2,D1.B)`

The destination operand must be a data register. Also the source operand is outside the range of signed values that can be represented in 8 bits.

3.27. Program trace:

| TIME              | D0  | D1 | A2   | N | NUM1 | SUM |
|-------------------|-----|----|------|---|------|-----|
| after 1st ADD.W   | 83  | 5  | 2402 | 5 | 2400 | 0   |
| after 2nd ADD.W   | 128 | 4  | 2404 | 5 | 2400 | 0   |
| after 3rd ADD.W   | 284 | 3  | 2406 | 5 | 2400 | 0   |
| after 4th ADD.W   | 34  | 2  | 2408 | 5 | 2400 | 0   |
| after 5th ADD.W   | 134 | 1  | 2410 | 5 | 2400 | 0   |
| after last MOVE.L | 134 | 0  | 2410 | 5 | 2400 | 134 |

- 3.28. (a) This program finds the location of the smallest element in a list whose starting address is stored in MEM1, and size in MEM2. The smallest element is stored in location DESIRED.
- (b) 16 words are required to store this program. We have assumed that the assembler uses short absolute addresses. (Long addresses are normally specified as MEM1.L, etc.) Otherwise, 3 more words would be needed.
- (c) The expression for memory accesses is  $T = 16 + 5n + 4m$ .
- 3.29. (a) They both leave the 17th negative word in RSLT.
- (b) Both programs scan through the list to find the 17th negative number in the list.
- (c) Program 1 takes 26 bytes of memory, while Program 2 requires 24.
- (d) Let  $P$  be the number of non-negative entries encountered. Program 1 requires  $9 + 7 \times 17 + 3 \times P$  and Program 2 requires  $10 + 6 \times 17 + 4 \times P$  memory accesses.
- (e) Program 1 requires slightly more memory, but has a clear speed advantage. Program 2 destroys the original list.
- 3.30. A 68000 program to compare two byte lists at locations X and Y, putting the larger byte at each position in a list starting at location LARGER, is:

|       |         |              |                                    |
|-------|---------|--------------|------------------------------------|
|       | MOVEA.L | #X,A0        |                                    |
|       | MOVEA.L | #Y,A1        |                                    |
|       | MOVEA.L | #LARGER,A2   |                                    |
|       | MOVE.W  | N,D0         |                                    |
|       | SUBQ    | #1,D0        | Initialize D0 to [N]−1             |
| LOOP  | CMP.B   | (A0)+,(A1)+  | Compare lists and advance pointers |
|       | BGT     | LISTY        |                                    |
|       | MOVE.B  | −1(A0),(A2)+ | Copy item from list X              |
|       | BRA     | NEXT         | Check next item                    |
| LISTY | MOVE.B  | −1(A1),(A2)+ | Copy item from list Y              |
| NEXT  | DBRA    | D0,LOOP      | Continue if more entries           |

3.31. A 68000 program for string matching:

|         |         |                         |                                  |
|---------|---------|-------------------------|----------------------------------|
|         | MOVEA.L | #STRING,A0              | Get location of STRING           |
|         | MOVE.W  | N,D0                    | Load D0 with appropriate         |
|         | MOVE.W  | M,D1                    | count for “match attempts”       |
|         | SUB.W   | D1,D0                   |                                  |
| LOOP    | MOVEA.L | #SUBSTRING,A1           | Get location of SUBSTRING        |
|         | MOVE.W  | M,D1                    | Get size of SUBSTRING            |
|         | MOVE.L  | A0,A2                   | Save location in STRING at which |
|         |         |                         | comparison will start            |
| MATCHER | DBRA    | D1,SUCCESS              |                                  |
|         | CMP.B   | (A0)+,(A1)+             | Compare and advance pointers     |
|         | BEQ     | MATCHER                 | If same, check next character    |
|         | MOVEA.L | A2,A0                   | Match failed; advance starting   |
|         | ADDQ.L  | #1,A0                   | character position in STRING     |
|         | DBRA    | D0,LOOP                 | Check if end of STRING           |
|         | MOVE.L  | #0,D0                   | Substring not found              |
|         | BRA     | NEXT                    |                                  |
| SUCCESS | MOVEA.L | A2,D0                   | Save location where match found  |
| NEXT    |         | <i>Next instruction</i> |                                  |

Note that DBRA is used in two ways in this program, once at the beginning and once at the end of a loop. In the first case, the counter is initialized to [M], while in the second the corresponding counter is initialized to [N]−[M]. This arrangement handles a substring of zero length correctly, and stops the attempt to find a match at the proper position.



3.32. A 68000 program to generate the first  $n$  numbers of the Fibonacci series:

|      |         |            |                                 |
|------|---------|------------|---------------------------------|
|      | MOVEA.L | #MEMLOC,A0 | Starting address                |
|      | MOVE.B  | N,D0       | Number of entries               |
|      | CLR     | D1         | The first entry = 0             |
|      | MOVE.B  | D1,(A0)+   |                                 |
|      | MOVE    | #1,D2      | The second entry = 1            |
|      | MOVE.B  | D2,(A0)+   |                                 |
|      | SUBQ.B  | #3,D0      | First two entries already saved |
| LOOP | MOVE.B  | -2(A0),D1  | Get second-last value           |
|      | ADD.B   | D1,D2      | Add to last value               |
|      | MOVE.B  | D2,(A0)+   | Store new value                 |
|      | DBRA    | D0,LOOP    |                                 |

The first 15 numbers in the Fibonacci sequence are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377. Therefore, the largest value of  $n$  that this program can handle is 14, because the largest number that can be stored in a byte is 255.

3.33. Let A0 point to the ASCII word. To change to uppercase, we need to change bit  $b_5$  from 1 to 0.

|      |                         |          |                           |
|------|-------------------------|----------|---------------------------|
| NEXT | MOVE.B                  | (A0),D0  | Get character             |
|      | CMP.B                   | #\$20,D0 | Check if space character  |
|      | BEQ                     | END      |                           |
|      | ANDI.B                  | #\$DF,D0 | Clear bit 5               |
|      | MOVE.B                  | D0,(A0)+ | Store converted character |
|      | BRA                     | NEXT     |                           |
| END  | <i>Next instruction</i> |          |                           |

- 3.34. Let  $\text{Stride} = 2(j + 1)$ , which is the distance in bytes between scores on a particular test for adjacent students in the list.

|       |         |          |                                                                     |
|-------|---------|----------|---------------------------------------------------------------------|
|       | MOVE    | J,D3     | Compute $\text{Stride} = 2(j + 1)$                                  |
|       | ADDQ    | #1,D3    |                                                                     |
|       | LSL     | #1,D3    |                                                                     |
|       | MOVEA.L | #SUM,A4  | Use A4 as pointer to the sums                                       |
|       | MOVEA.L | #LIST,A5 | Use A5 as pointer to scores                                         |
|       | ADDQ    | #2,A5    | for student 1                                                       |
|       | MOVE    | J,D6     | Use D6 as outer loop counter                                        |
|       | SUBQ    | #1,D6    | Adjust for use of DBRA instruction                                  |
| OUTER | MOVE    | N,D7     | Use D7 as inner loop counter                                        |
|       | SUBQ    | #1,D7    | Adjust for use of DBRA instruction                                  |
|       | MOVE    | A5,A2    | Use A2 as base for scanning test scores                             |
|       | CLR     | D0       | Use D0 as sum accumulator                                           |
| INNER | ADD     | [A2],D0  | Accumulate test scores                                              |
|       | ADD     | D3,A2    | Point to next score                                                 |
|       | DBRA    | D7,INNER | Check if score for current test<br>for all students have been added |
|       | MOVE    | D0,[A4]  | Store sum in memory                                                 |
|       | ADDQ    | #2,A5    | Increment to next test                                              |
|       | ADDQ    | #2,A4    | Point to next sum                                                   |
|       | DBRA    | D6,OUTER | Check if scores for all tests<br>have been accumulated              |

- 3.35. This program is similar to Figure 3.27, and makes the same assumptions about status word bit locations.

|      |        |              |                          |
|------|--------|--------------|--------------------------|
|      | MOVE   | #N,D0        |                          |
|      | SUBQ.W | #1,D0        | Initialize D0 to $n - 1$ |
| READ | BTST.W | #3,INSTATUS  |                          |
|      | BEQ    | READ         | Wait for data ready      |
|      | MOVE.B | DATAIN,D1    | Get new character        |
|      | MOVE.B | D1,−(A0)     | Push on user stack       |
| ECHO | BTST.W | #3,OUTSTATUS |                          |
|      | BEQ    | ECHO         | Wait for terminal ready  |
|      | MOVE.B | D1,DATAOUT   | Output new character     |
|      | DBRA   | D0,READ      | Read next character      |

- 3.36. Assume that most of the time between successive characters being struck is spent in the two-instruction wait loop that starts at location READ. The BEQ READ instruction is executed once every 40 ns while this loop is being executed. There are  $10^9/10 = 10^8$  ns between successive characters. Therefore, the BEQ READ instruction is executed  $10^8/40 = 2.5 \times 10^6$  times per character entered.
- 3.37. Assume that register A4 is used as a memory pointer by the main program.

#### Main Program

|          |        |          |                                    |
|----------|--------|----------|------------------------------------|
| READLINE | BSR    | GETCHAR  | Call character read subroutine.    |
|          | MOVE.B | D0,(A4)+ | Store character in memory.         |
|          | BSR    | PUTCHAR  | Call character display subroutine. |
|          | CMPL.B | #CR,D0   | Check for end-of-line character.   |
|          | BNE    | READLINE |                                    |

#### Subroutine GETCHAR

|         |        |         |                         |
|---------|--------|---------|-------------------------|
| GETCHAR | BTST.W | #3,(A0) | Wait for character.     |
|         | BEQ    | GETCHAR |                         |
|         | MOVE.B | (A1),D0 | Load character into D0. |
|         | RTS    |         | Return.                 |

#### Subroutine PUTCHAR

|         |        |         |                            |
|---------|--------|---------|----------------------------|
| PUTCHAR | BTST.W | #3,(A2) | Wait for display.          |
|         | BEQ    | PUTCHAR |                            |
|         | MOVE.B | D0,(A3) | Send character to display. |
|         | RTS    |         | Return.                    |

3.38. Addresses INSTATUS and DATAIN are pushed onto the processor stack in that order by the main program as parameters for GETCHAR. The character read is passed back to the main program in the DATAIN position on the stack. The addresses OUTSTATUS and DATAOUT and the character to be displayed are pushed onto the processor stack in that order by the main program as parameters for PUTCHAR. A stack structure like that shown in Figure 3.29 is used.

GETCHAR uses registers A0, A1, and D0 to hold INSTATUS, DATAIN, and the character read.

PUTCHAR uses registers A0, A1, and D0 to hold OUTSTATUS, DATAOUT, and the character to be displayed.

The main program uses register A0 as a memory pointer, and uses register D0 to hold the character read.

### Main Program

|          |        |                   |                                     |
|----------|--------|-------------------|-------------------------------------|
| READLINE | MOVE.L | #INSTATUS, -(A7)  | Push address parameters             |
|          | MOVE.L | #DATAIN, -(A7)    | onto the stack.                     |
|          | BSR    | GETCHAR           | Call character read subroutine.     |
|          | MOVE.L | (A7)+, D0         | Pop long word containing            |
|          | MOVE.B | D0, (A0)+         | character from top of               |
|          |        |                   | stack into D0 and                   |
|          |        |                   | store character into memory.        |
|          | ADDI   | #4, A7            | Remove INSTATUS from stack.         |
|          | MOVE.L | #OUTSTATUS, -(A7) | Push address parameters             |
|          | MOVE.L | #DATAOUT, -(A7)   | onto stack.                         |
|          | MOVE.L | D0, -(A7)         | Push long word containing           |
|          |        |                   | character onto stack.               |
|          | BSR    | PUTCHAR           | Call character display subroutine.  |
|          | ADDI   | #12, A7           | Remove three parameters from stack. |
|          | CMPL.B | #CR, D0           | Check for end-of-line character.    |
|          | BNE    | READLINE          |                                     |

### Subroutine GETCHAR

|         |        |                 |                                |
|---------|--------|-----------------|--------------------------------|
| GETCHAR | MOVEM  | D0/A0-A1, -(A7) | Save registers.                |
|         | MOVE.L | 20(A7), A0      | Load address INSTATUS into A0. |
|         | MOVE.L | 16(A7), A1      | Load address DATAIN into A1.   |
| READ    | BTST   | #3, (A0)        | Wait for character.            |
|         | BEQ    | READ            |                                |
|         | MOVE.B | (A1), D0        | Load character into D0 and     |
|         | MOVE.L | D0, 16(A7)      | push onto the stack,           |
|         |        |                 | overwriting DATAIN.            |
|         | MOVEM  | (A7)+, D0/A0-A1 | Restore registers.             |
|         | RTS    |                 | Return.                        |

### Subroutine PUTCHAR

|         |        |                 |                                                 |
|---------|--------|-----------------|-------------------------------------------------|
| PUTCHAR | MOVEM  | D0/A0-A1, -(A7) | Save registers.                                 |
|         | MOVE.L | 24(A7), A0      | Load address OUTSTATUS into A0.                 |
|         | MOVE.L | 20(A7), A1      | Load address DATAOUT into A1.                   |
|         | MOVE.L | 16(A7), D0      | Load long word containing<br>character into D0. |
| DISPLAY | BTST   | #3, (A0)        | Wait for device ready.                          |
|         | BEQ    | DISPLAY         |                                                 |
|         | MOVE.B | D0, (A1)        | Send character to display.                      |
|         | MOVEM  | (A7)+, D0/A0-A1 | Restore registers.                              |
|         | RTS    |                 | Return.                                         |

- 3.39. See the solution to Problem 2.18 for the procedures needed to perform the append and remove operations.

Register assignment:

D0 – Data byte to append to or remove from queue  
A1 – IN pointer  
A2 – OUT pointer  
A3 – Address of first queue byte location  
A4 – Address of last queue byte location ( $= [A3] + k - 1$ )  
A5 – Auxiliary register for address of next appended byte

Initially, the queue is empty with  $[A1] = [A2] = [A3]$

APPEND routine:

|        |         |           |                                 |
|--------|---------|-----------|---------------------------------|
|        | MOVEA.L | A1,A5     |                                 |
|        | ADDQ.L  | #1,A1     | Increment A1 Modulo $k$ .       |
|        | CMPA.L  | A1,A4     |                                 |
|        | BGE     | CHECK     |                                 |
|        | MOVEA.L | A3,A1     |                                 |
| CHECK  | CMPA.L  | A1,A2     | Check if queue is full.         |
|        | BNE     | APPEND    | If queue not full, append byte. |
|        | MOVEA.L | A5,A1     | Otherwise, restore              |
|        | BRA     | QUEUEFULL | IN pointer and send             |
|        |         |           | message that queue is full.     |
| APPEND | MOVE.B  | D0,[A5]   | Append byte.                    |

REMOVE routine:

|      |         |            |                          |
|------|---------|------------|--------------------------|
|      | CMPA.L  | A1,A2      | Check if queue is empty. |
|      | BEQ     | QUEUEEMPTY | If empty, send message.  |
|      | MOVE.B  | (A2)+,D0   | Otherwise, remove byte   |
|      | CMPA.L  | A2,A4      | and increment A2         |
|      | BGE     | NEXT       | Modulo $k$ .             |
|      | MOVEA.L | A3,A2      |                          |
| NEXT | ...     |            |                          |

3.40. Using the same assumptions as in Problem 3.35 and its solution, a 68000 program to convert 3 input decimal digits to a binary number is:

|        |                    |                                       |
|--------|--------------------|---------------------------------------|
| BSR    | READ               | Get first character                   |
| ASL    | #1,D0              | Multiply by 2 for word offset         |
| MOVE.W | HUNDREDS(D0),D1    | Get hundreds value                    |
| BSR    | READ               | Get second character                  |
| ASL    | #1,D0              | Multiply by 2 for word offset         |
| ADD.W  | TENS(D0),D1        | Get tens value                        |
| BSR    | READ               | Get third character                   |
| ADD.W  | D0,D1              | D1 contains value of binary<br>number |
| READ   | BTST.W #3,INSTATUS |                                       |
|        | BEQ READ           | Wait for new character                |
|        | MOVE.B DATAIN,D0   | Get new character                     |
|        | AND.B #\$0F,D0     | Convert to equivalent binary<br>value |
|        | RTS                |                                       |

3.41. (a) The subroutines convert 3 decimal digits to a binary value.

|            |         |                 |                                      |
|------------|---------|-----------------|--------------------------------------|
| GETDECIMAL | MOVEM.L | D0/A0–A1,–(A7)  | Save registers                       |
|            | MOVEA.L | 20(A7),A0       | Get string buffer address            |
|            | MOVE.B  | #2,D0           | Use D0 as character counter          |
|            | BTST.W  | #3,INSTATUS     |                                      |
|            | BEQ     | READ            |                                      |
|            | MOVE.B  | DATAIN,(A0)+    | Get and store character              |
|            | DBRA    | D0,READ         | Repeat until all characters received |
|            | MOVE.L  | 16(A7),A1       | Pointer to result                    |
|            | BSR     | CONVERSION      |                                      |
|            | MOVEM.L | (A7)+,D0/A0–A1  | Restore registers                    |
| CONVERSION | RTS     |                 |                                      |
|            | MOVEM.L | D0–D1,–(A7)     | Save registers                       |
|            | MOVE.B  | –(A0),D0        | Get least sig. digit                 |
|            | AND.W   | #\$0F,D0        | Numeric value of digit               |
|            | MOVE.B  | –(A0),D1        | Get tens digit                       |
|            | AND.W   | #\$0F,D1        | Numeric value of digit               |
|            | ASL     | #1,D1           |                                      |
|            | ADD.W   | TENS(D1),D0     | Add tens value                       |
|            | MOVE.B  | –(A0),D1        | Get hundreds digit                   |
|            | AND.W   | #\$0F,D1        | Numeric value of digit               |
|            | ASL     | #1,D1           |                                      |
|            | ADD.W   | HUNDREDS(D1),D0 | Add hundreds value                   |
|            | MOVE.W  | D0,(A1)         | Store result                         |
|            | MOVEM.L | (A7)+,D0–D1     | Restore registers                    |
|            | RTS     |                 |                                      |

(b) The contents of the top of the stack after the call to the CONVERSION routine are:

|                              |
|------------------------------|
|                              |
| Return address of CONVERSION |
| D0 <sub>MAIN</sub>           |
| A1 <sub>MAIN</sub>           |
| A0 <sub>MAIN</sub>           |
| Return address of GETDECIMAL |
| Result address               |
| Buffer address               |
| ORIG TOS                     |
|                              |



- 3.42. Assume that the subroutine can change the contents of any registers used to pass parameters. Let  $\text{Stride} = 2m$ , which is the distance in bytes between successive word elements in a given column.

|       |      |            |                                                  |
|-------|------|------------|--------------------------------------------------|
|       | LSL  | #1,D4      | Set Stride in D4                                 |
|       | SUB  | D1,D2      | Set D2 to contain                                |
|       | LSL  | #1,D2      | $2(y - x)$                                       |
|       | LSL  | #1,D1      | Set A0 to address                                |
|       | ADDA | D1,A0      | $A(0,x)$                                         |
|       | BRA  | START      |                                                  |
| LOOP  | MOVE | (A0),D1    | Load $[A(i,x)]$ into D1                          |
|       | ADD  | D1,(A0,D2) | Add array elements                               |
|       | ADD  | D4,A0      | Move to next row                                 |
| START | DBRA | D3,LOOP    | Repeat loop until all<br>entries have been added |
|       | RTS  |            | Return                                           |

Note that LOOP is entered by branching to the DBRA instruction. So DBRA decrements D3 to contain  $n - 1$ , which is the correct starting value when the DBRA instruction is used.

- 3.43. A 68000 program to reverse the order of bits in register D2:

|      |      |         |                              |
|------|------|---------|------------------------------|
|      | MOVE | #15,D0  | Use D0 as counter            |
|      | CLR  | D1      | D1 will receive new value    |
| LOOP | LSL  | D2      | Shift MSB of D2 into X bit   |
|      | ROXR | D1      | Shift X bit into MSB of D1   |
|      | DBRA | D0,LOOP | Repeat until D0 reaches $-1$ |
|      | MOVE | D1,D2   | Put new value back in D2     |

|       |                  |              |
|-------|------------------|--------------|
| 3.44. |                  | Bytes/access |
|       | MOVEA.L #LOC,A0  | 6/3          |
|       | MOVE.B (A0)+,D0  | 2/2          |
|       | LSL.B #4,D0      | 2/1          |
|       | MOVE.B (A0),D1   | 2/2          |
|       | ANDI.B #\$F,D1   | 4/2          |
|       | OR.B D0,D1       | 2/1          |
|       | MOVE.B D1,PACKED | 4/3          |

Total size is 22 bytes and execution involves 14 memory access cycles.

3.45. The trace table is:

| TIME                | 1000 | 1001 | 1002 | 1003 | 1004 | D1 | D2 | D3  |
|---------------------|------|------|------|------|------|----|----|-----|
| after 1st BGT OUTER | 106  | 13   | 67   | 45   | 120  | 3  | -1 | 120 |
| after 2nd BGT OUTER | 67   | 13   | 45   | 106  | 120  | 2  | -1 | 106 |
| after 3rd BGT OUTER | 45   | 13   | 67   | 106  | 120  | 1  | -1 | 67  |
| after 4th BGT OUTER | 13   | 45   | 67   | 106  | 120  | 0  | -1 | 45  |

3.46. Assume the list address is passed to the subroutine in register A1. When the subroutine is entered, the number of list entries needs to be loaded into D1. Then A1 must be updated to point to the first entry in the list. Because addresses must be incremented or decremented by 2 to handle word quantities, the address mode (A1,D1) is no longer useful. Also, since the initial address points to the beginning of the list, we will scan the list forwards.

|       |       |           |                                                              |
|-------|-------|-----------|--------------------------------------------------------------|
|       | MOVE  | (A1)+,D1  | Load number of entries, $n$                                  |
|       | SUBQ  | #2,D1     | Outer loop counter $\leftarrow n - 2$ ( $j$ : 0 to $n - 2$ ) |
| OUTER | MOVE  | D1,D2     | Inner loop $\leftarrow$ outer loop counter                   |
|       | MOVEA | A1,A2     | Use A2 as a pointer in the inner loop                        |
|       | ADDQ  | #2,A2     | $k \leftarrow j + 1$ ( $k$ : 1 to $n - 1$ )                  |
| INNER | MOVE  | (A1),D3   | Current maximum value in D3                                  |
|       | CMP   | (A2),D3   |                                                              |
|       | BLE   | NEXT      | If $LIST(j) \leq LIST(k)$ , go to next                       |
|       | MOVE  | (A2),(A1) | Interchange $LIST(k)$                                        |
|       | MOVE  | D3,(A2)   | and $LIST(j)$ .                                              |
| NEXT  | ADDQ  | #2,A2     |                                                              |
|       | DBRA  | D2,INNER  |                                                              |
|       | ADDQ  | #2,A1     |                                                              |
|       | DBRA  | D1,OUTER  | If not finished,                                             |
|       | RTS   |           | return                                                       |

- 3.47. Use D4 to keep track of the position of the largest element in the inner loop and D5 to record its value.

|       |         |                 |                                             |
|-------|---------|-----------------|---------------------------------------------|
|       | MOVEA.L | #LIST,A1        | Pointer to the start of the list            |
|       | MOVE    | N,D1            | Initialize outer loop                       |
|       | SUBQ    | #1,D1           | index j in D <sub>1</sub>                   |
| OUTER | MOVE    | D1,D2           | Initialize inner loop                       |
|       | SUBQ    | #1,D2           | index k in D <sub>2</sub>                   |
|       | MOVE.L  | D1,D4           | Index of largest element                    |
|       | MOVE.B  | (A1,D1),D5      | Value of largest element                    |
| INNER | MOVE.B  | (A1,D2),D3      | Get new element, LIST( <i>k</i> )           |
|       | CMP.B   | D3,D5           | Compare to current maximum                  |
|       | BCC     | NEXT            | If lower go to next entry                   |
|       | MOVE.L  | D2,D4           | Update index of largest element             |
|       | MOVE.L  | D3,D5           | Update largest value                        |
| NEXT  | DBRA    | D2,INNER        | Inner loop control                          |
|       | MOVE.B  | (A1,D1),(A1,D4) | Swap LIST( <i>j</i> ) and LIST( <i>k</i> ); |
|       | MOVE.B  | D5,(A1,D1)      | correct even if same                        |
|       | SUBQ    | #1,D1           | Branch back                                 |
|       | BGT     | OUTER           | if not finished                             |

The potential advantage is that the inner loop of the new program should execute faster.

- 3.48. Assume that register A0 points to the first record. We will use registers D1, D2, and D3 to accumulate the three sums. Assume also that the list is not empty.

|      |         |           |                              |
|------|---------|-----------|------------------------------|
|      | CLR     | D1        |                              |
|      | CLR     | D2        |                              |
|      | CLR     | D3        |                              |
| LOOP | ADD.L   | 8(A0),D1  | Accumulate scores for test 1 |
|      | ADD.L   | 12(A0),D2 | Accumulate scores for test 2 |
|      | ADD.L   | 16(A0),D3 | Accumulate scores for test 3 |
|      | MOVE.L  | 4(A0),D0  | Get link                     |
|      | MOVEA.L | D0,A0     | Load in pointer register     |
|      | BNE     | LOOP      |                              |
|      | MOVE.L  | D1,SUM1   |                              |
|      | MOVE.L  | D2,SUM2   |                              |
|      | MOVE.L  | D3,SUM3   |                              |

Note that the MOVE instruction that reads the link value into register D0 sets the Z and N flags. The MOVEA instruction does not affect the condition code flags. Hence, the BNE instruction will test the correct values.

- 3.49. In the program of Figure 3.35, if the ID of the new record matches the ID of the Head record, the new record will become the new Head. If the ID matches that of a later record, it will be inserted immediately after that record, including the case where the matching record is the Tail.

Modify the program as follows.

|                                            |                |                                             |
|--------------------------------------------|----------------|---------------------------------------------|
| Add the following as the first instruction |                |                                             |
| INSERTION                                  | MOVE.L #0,A6   | Anticipate a successful insertion           |
| After the instruction labeled HEAD insert  |                |                                             |
|                                            | BEQ DUPLICATE1 | New record matches head                     |
| After the BLT INSERT instruction insert    |                |                                             |
|                                            | BEQ DUPLICATE2 | New record matches a record other than head |
| Add the following instructions at the end  |                |                                             |
| DUPLICATE1                                 | MOVE.L A0,A6   | Return the address of the head              |
|                                            | RTS            |                                             |
| DUPLICATE2                                 | MOVE.L A3,A6   | Return address of matching record           |
|                                            | RTS            |                                             |

- 3.50. If the ID of the new record is less than that of the head, the program in Figure 3.36 will delete the head. If the list is empty, the result is unpredictable because the first instruction compares the new ID with the contents of memory location zero. If the list is not empty, the program continues until A2 points to the Tail record. Then the instruction at LOOP loads zero into A3 and the result is unpredictable.

To correct behavior, modify the program as follows.

|                                            |              |                                                                                           |
|--------------------------------------------|--------------|-------------------------------------------------------------------------------------------|
| After the first BGT instruction insert     |              |                                                                                           |
|                                            | BLT ERROR    | ID of new record less than head                                                           |
|                                            | MOVE.L #0,D1 | Deletion successful                                                                       |
| After the BEQ DELETE instruction insert    |              |                                                                                           |
|                                            | BGT ERROR    | ID of New record is less than that of the next record and greater than the current record |
| Add the following instruction after DELETE |              |                                                                                           |
|                                            | MOVE.L #0,D1 | Deletion successful                                                                       |
| Add the following instruction at the end   |              |                                                                                           |
| ERROR                                      | RTS          | Record not in the list                                                                    |

## PART III: Intel IA-32

3.51. Initial memory contents are:

$[1000] = 1$   
 $[1004] = 2$   
 $[1008] = 3$   
 $[1012] = 4$   
 $[1016] = 5$   
 $[1020] = 6$

(a)  $[EBX + ESI*4 + 8] = 1016$

$EAX \leftarrow 10 + 5 = 15$

(b) The values 20 and 30 are pushed onto the processor stack, and then 30 is popped into EAX and 20 is popped into EBX. The Subtract instruction then performs  $30 - 20$ , and places the result of 10 into EAX.

(c) The address value 1008 is loaded into EAX, and then 3 is loaded into EBX.

3.52. (a) OK

(b) ERROR: Only one operand can be in memory.

(c) OK

(d) ERROR: Scale factor can only be 1, 2, 4, or 8.

(e) OK

(f) ERROR: An immediate operand can not be a destination.

(g) ERROR: ESP cannot be used as an index register.

3.53. Program trace:

| TIME                        | EAX  | EBX      | ECX |
|-----------------------------|------|----------|-----|
| After 1st execution of LOOP | -113 | NUM1 - 4 | 4   |
| After 2nd execution of LOOP | 129  | NUM1 - 4 | 3   |
| After 3rd execution of LOOP | 78   | NUM1 - 4 | 2   |

3.54. Assume bytes are unsigned 8-bit values.

|         |      |                 |                                     |
|---------|------|-----------------|-------------------------------------|
|         | MOV  | ECX,N           | ECX is list counter.                |
|         | LEA  | ESI,X           | ESI points to X list.               |
|         | SUB  | ESI,1           |                                     |
|         | LEA  | EDI,Y           | EDI points to Y list.               |
|         | SUB  | EDI,1           |                                     |
|         | LEA  | EDX,LARGER      | EDX points to LARGER list.          |
|         | SUB  | EDX,1           |                                     |
| START:  | MOV  | AL,[ESI + ECX]  | Load X byte into AL.                |
|         | MOV  | BL,[EDI + ECX], | Load Y byte into BL.                |
|         | CMP  | AL,BL           | Compare bytes.                      |
|         | JAE  | XLARGER         | Branch if X byte<br>larger or same. |
|         | MOV  | [EDX + ECX],BL  | Otherwise, store<br>Y byte.         |
|         | JMP  | CHECK           |                                     |
| XLARGER | MOV  | [EDX + ECX],AL  | Store X byte.                       |
| CHECK   | LOOP | START           | Check if done.                      |

3.55. The inner loop checks for a match at each possible position.

|          |      |               |                                |
|----------|------|---------------|--------------------------------|
|          | MOV  | EDX,N         | Compute outer loop count       |
|          | SUB  | EDX,M         | and store in EDX.              |
|          | INC  | EDX           |                                |
|          | LEA  | EAX,STRING    | Use EAX as a base              |
|          |      |               | pointer for each match         |
|          |      |               | attempt.                       |
| OUTER:   | MOV  | ESI,EAX       | Use ESI and EDI as             |
|          | LEA  | EDI,SUBSTRING | running pointers for           |
|          |      |               | each match attempt.            |
|          | MOV  | ECX,M         | Initialize inner loop counter. |
| INNER:   | MOV  | BL,[EDI]      | Load next substring byte       |
|          | CMP  | BL,[ESI]      | into BL and compare to         |
|          |      |               | corresponding string byte.     |
|          | JNE  | NOMATCH       | If not equal, go to            |
|          |      |               | next substring position.       |
|          | INC  | ESI           | If equal, increment running    |
|          | INC  | EDI           | pointers to next byte          |
|          |      |               | positions.                     |
|          | LOOP | INNER         | Check if all substring         |
|          |      |               | bytes compared.                |
|          | JMP  | NEXT          | If a match is found,           |
|          |      |               | exit with string position      |
|          |      |               | in EAX.                        |
| NOMATCH: | INC  | EAX           | Increment EAX to next possible |
|          |      |               | substring position.            |
|          | DEC  | EDX           | Check if all positions tried.  |
|          | JG   | OUTER         |                                |
|          | MOV  | EAX,0         | If yes, load zero into         |
|          |      |               | EAX and exit.                  |
| NEXT:    | ...  |               |                                |

- 3.56. This solution assumes that the last number in the series of  $n$  numbers can be represented in a 32-bit doubleword, and that  $n > 2$ .

|            |      |               |                           |
|------------|------|---------------|---------------------------|
|            | MOV  | ECX,N         | Use ECX to count numbers  |
|            | SUB  | ECX,2         | generated after 1.        |
|            | LEA  | EDI,MEMLOC    | Use EDI as a memory       |
|            |      |               | pointer.                  |
|            | MOV  | EAX,0         | Store first two numbers   |
|            | MOV  | [EDI],EAX     | from EAX and EBX into     |
|            | MOV  | EBX,1         | memory.                   |
|            | ADD  | EDI,4         |                           |
|            | MOV  | [EDI],EBX     |                           |
| LOOPSTART: | ADD  | EDI,4         | Increment memory pointer. |
|            | MOV  | EAX,[EDI - 8] | Load second last value.   |
|            | ADD  | EBX,EAX       | Add to last value.        |
|            | MOV  | [EDI],EBX     | Store new value.          |
|            | LOOP | LOOPSTART     | Check if all $n$ numbers  |
|            |      |               | generated.                |

- 3.57. Assume register EAX contains the address (WORD) of the first character. To change characters from lowercase to uppercase, change bit  $b_5$  from 1 to 0.

|       |     |          |                              |
|-------|-----|----------|------------------------------|
| NEXT: | MOV | BL,[EAX] | Load next character into BL. |
|       | CMP | BL,20H   | Check if space character.    |
|       | JE  | END      | If space, exit.              |
|       | AND | BL,DFH   | Clear bit $b_5$ .            |
|       | MOV | [EAX],BL | Store converted character.   |
|       | INC | EAX      | Increment memory pointer.    |
|       | JMP | NEXT     | Convert next character.      |
| END:  | ... |          |                              |



- 3.58. The parameter  $\text{Stride} = (j + 1)$  is the distance in doublewords between scores on a particular test for adjacent students in the list.

|        |      |                   |                                                                  |
|--------|------|-------------------|------------------------------------------------------------------|
|        | MOV  | EDX,J             | Load outer loop counter EDX.                                     |
|        | INC  | J                 | Increment memory location J to contain $\text{Stride} = j + 1$ . |
|        | LEA  | EBX,SUM           | Load address SUM into EBX.                                       |
|        | LEA  | EDI,LIST          | Load address of test score 1 for student 1 into EDI.             |
| OUTER: | ADD  | EDI,4             |                                                                  |
|        | MOV  | ECX,N             | Load inner loop counter ECX.                                     |
|        | MOV  | EAX,0             | Clear scores accumulator EAX.                                    |
|        | MOV  | ESI,0             | Clear index register ESI.                                        |
| INNER: | ADD  | EAX,[EDI + ESI*4] | Add next test score.                                             |
|        | ADD  | ESI,J             | Increment index register ESI by Stride value.                    |
|        | LOOP | INNER             | Check if all $n$ scores have been added.                         |
|        | MOV  | [EBX],EAX         | Store current test sum.                                          |
|        | ADD  | EBX,4             | Increment sum location pointer.                                  |
|        | ADD  | EDI,4             | Increment base pointer to next test score for student 1.         |
|        | DEC  | EDX               | Check if all test scores summed.                                 |
|        | JG   | OUTER             |                                                                  |

This solution uses six of the IA-32 registers. It does not use registers EBP or ESP, which are normally reserved as pointers for the processor stack. Use of EBP to hold the parameter Stride would result in a somewhat more efficient inner loop.

- 3.59. Use register ECX as a counter register, and use EBX as a work register.

|            |      |           |                                                                                          |
|------------|------|-----------|------------------------------------------------------------------------------------------|
|            | MOV  | ECX,32    | Load ECX with count value 32.                                                            |
|            | MOV  | EBX,0     | Clear work register EBX.                                                                 |
| LOOPSTART: | SHL  | EAX,1     | Shift contents of EAX left one bit position, moving the high-order bit into the CF flag. |
|            | RCR  | EBX,1     | Rotate EBX right one bit position, including the CF flag.                                |
|            | LOOP | LOOPSTART | Check if finished.                                                                       |
|            | MOV  | EAX,EBX   | Load reversed pattern into EAX.                                                          |

- 3.60. See the solution to Problem 2.18 for the procedures needed to perform the append and remove operations.

Register assignment:

AL — Data byte to append to or remove from the queue  
 ESI — IN pointer  
 EDI — OUT pointer  
 EBX — Address of first queue byte location  
 ECX — Address of last queue byte location (  $[EBX] + k - 1$  )  
 EDX — Auxiliary register for location of next appended byte

Initially, the queue is empty with  $[ESI] = [EDI] = [EBX]$ .

Append routine:

|         |     |           |                                                                       |
|---------|-----|-----------|-----------------------------------------------------------------------|
|         | MOV | EDX,ESI   | Save current value of IN<br>pointer ESI in auxiliary<br>register EDX. |
|         | INC | ESI       | These four instructions<br>increment ESI Modulo $k$ .                 |
|         | CMP | ECX,ESI   |                                                                       |
|         | JGE | CHECK     |                                                                       |
|         | MOV | ESI,EBX   |                                                                       |
| CHECK:  | CMP | EDI,ESI   | Check if queue is full.                                               |
|         | JNE | APPEND    | If not full, append byte.                                             |
|         | MOV | ESI,EDX   | Otherwise, restore IN pointer                                         |
|         | JMP | QUEUEFULL | and send message that<br>queue is full.                               |
| APPEND: | MOV | [EDX],AL  | Append byte.                                                          |

Remove routine:

|       |     |            |                            |
|-------|-----|------------|----------------------------|
|       | CMP | EDI,ESI    | Check if queue is empty.   |
|       | JE  | QUEUEEMPTY | If empty, send message.    |
|       | MOV | AL,[EDI]   | Otherwise, remove byte and |
|       | INC | EDI        | increment EDI Modulo $k$ . |
|       | CMP | ECX,EDI    |                            |
|       | JGE | NEXT       |                            |
|       | MOV | EDI,EBX    |                            |
| NEXT: | ... |            |                            |

- 3.61. This program is similar to Figure 3.44; and it makes the same assumptions about status word bit locations.

|       |      |             |                                   |
|-------|------|-------------|-----------------------------------|
|       | MOV  | ECX,N       | Use ECX as the loop counter.      |
| READ: | BT   | INSTATUS,3  | Wait for the character.           |
|       | JNC  | READ        |                                   |
|       | MOV  | AL,DATAIN   | Transfer character into AL.       |
|       | DEC  | EBX         | Push character onto user stack.   |
|       | MOV  | [EBX],AL    |                                   |
| ECHO: | BT   | OUTSTATUS,3 | Wait for the display.             |
|       | JNC  | ECHO        |                                   |
|       | MOV  | DATAOUT,AL  | Send character to display.        |
|       | LOOP | READ        | Check if all $n$ characters read. |

- 3.62. Assume that most of the time between successive characters being struck is spent in the two-instruction wait loop that starts at location READ. The JNC READ instruction is executed once every 20 ns while this loop is being executed. There are  $10^9/10 = 10^8$  ns between successive characters. Therefore, the JNC READ instruction is executed  $10^8/20 = 5 \times 10^6$  times per character entered.

- 3.63 Assume that register ECX is used as a memory pointer by the main program.

#### Main Program

|           |      |          |                            |
|-----------|------|----------|----------------------------|
| READLINE: | CALL | GETCHAR  |                            |
|           | MOV  | [ECX],AL | Store character in memory. |
|           | INC  | ECX      | Increment memory pointer.  |
|           | CALL | PUTCHAR  |                            |
|           | CMP  | AL,CR    | Check for end-of-line.     |
|           | JNE  | READLINE | Go back for more.          |

#### Subroutine GETCHAR

|          |     |                   |                         |
|----------|-----|-------------------|-------------------------|
| GETCHAR: | BT  | DWORD PTR [EBX],3 | Wait for character.     |
|          | JNC | GETCHAR           |                         |
|          | MOV | AL,[EDX]          | Load character into AL. |
|          | RET |                   |                         |

#### Subroutine PUTCHAR

|          |     |                   |                    |
|----------|-----|-------------------|--------------------|
| PUTCHAR: | BT  | DWORD PTR [ESI],3 | Wait for display.  |
|          | JNC | PUTCHAR           |                    |
|          | MOV | [EDI],AL          | Display character. |
|          | RET |                   |                    |

3.64. Addresses INSTATUS and DATAIN are pushed onto the processor stack in that order by the main program as parameters for GETCHAR. The character read is passed back to the main program in the DATAIN position on the stack. The addresses OUTSTATUS and DATAOUT and the character to be displayed are pushed onto the processor stack in that order by the main program as parameters for PUTCHAR. A stack structure like that shown in Figure 3.46 is used.

GETCHAR uses registers EBX, EDX, and AL (EAX) to hold INSTATUS, DATAIN, and the character read.

PUTCHAR uses registers ESI, EDI, and AL (EAX) to hold OUTSTATUS, DATAOUT, and the character to be displayed.

Assume that register ECX is used as a memory pointer by the main program.

#### Main Program

|           |      |                  |                               |
|-----------|------|------------------|-------------------------------|
| READLINE: | PUSH | OFFSET INSTATUS  | Push address parameters       |
|           | PUSH | OFFSET DATAIN    | onto the stack.               |
|           | CALL | GETCHAR          |                               |
|           | POP  | EAX              | Pop the doubleword            |
|           |      |                  | containing the character      |
|           |      |                  | read into EAX.                |
|           | MOV  | [ECX],AL         | Store character in            |
|           |      |                  | low-order byte of EAX         |
|           |      |                  | into the memory.              |
|           | INC  | ECX              | Increment the memory pointer. |
|           | ADD  | ESP,4            | Remove parameter INSTATUS     |
|           |      |                  | from top of the stack.        |
|           | PUSH | OFFSET OUTSTATUS | Push address parameters       |
|           | PUSH | OFFSET DATAOUT   | onto the stack.               |
|           | PUSH | EAX              | Push doubleword containing    |
|           |      |                  | the character to be displayed |
|           |      |                  | onto the stack.               |
|           | CALL | PUTCHAR          |                               |
|           | ADD  | ESP,12           | Remove three parameters       |
|           |      |                  | from the stack.               |
|           | CMP  | AL,CR            | Check for end-of-line         |
|           |      |                  | character.                    |
|           | JNE  | READLINE         | Go back for more.             |

### Subroutine GETCHAR

|          |      |                   |                                |
|----------|------|-------------------|--------------------------------|
| GETCHAR: | PUSH | EAX               | Save registers to be           |
|          | PUSH | EBX               | used in the subroutine.        |
|          | PUSH | EDX               |                                |
|          | MOV  | EBX,[ESP + 20]    | Load INSTATUS into EBX.        |
|          | MOV  | EDX,[ESP + 16]    | Load DATAIN into EDX.          |
| READ:    | BT   | DWORD PTR [EBX],3 | Wait for character.            |
|          | JNC  | READ              |                                |
|          | MOV  | AL,[EDX]          | Read character into AL.        |
|          | MOV  | [ESP + 16],EAX    | Overwrite DATAIN in the        |
|          |      |                   | stack with the doubleword      |
|          |      |                   | containing the character read. |
|          | POP  | EDX               | Restore registers.             |
|          | POP  | EBX               |                                |
|          | POP  | EAX               |                                |
|          | RET  |                   |                                |

### Subroutine PUTCHAR

|          |      |                   |                            |
|----------|------|-------------------|----------------------------|
| PUTCHAR: | PUSH | EAX               | Save registers to be       |
|          | PUSH | ESI               | used in the subroutine.    |
|          | PUSH | EDI               |                            |
|          | MOV  | ESI,[ESP + 24]    | Load OUTSTATUS.            |
|          | MOV  | EDI,[ESP + 20]    | Load DATAOUT.              |
|          | MOV  | EAX,[ESP + 16]    | Load doubleword containing |
|          |      |                   | character to be displayed  |
|          |      |                   | into register EAX.         |
| DISPLAY: | BT   | DWORD PTR [ESI],3 | Wait for the display.      |
|          | JNC  | DISPLAY           |                            |
|          | MOV  | [EDI],AL          | Display character.         |
|          | POP  | EDI               | Restore registers.         |
|          | POP  | ESI               |                            |
|          | POP  | EAX               |                            |
|          | RET  |                   |                            |

3.65. Using the same assumptions as in Problem 3.61 and its solution, an IA-32 program to convert 3 input decimal digits to a binary number is:

|       |      |                          |                                        |
|-------|------|--------------------------|----------------------------------------|
|       | CALL | READ                     | Get first character                    |
|       | MOV  | EBX,[HUNDREDS + EAX * 4] | Get hundreds value                     |
|       | CALL | READ                     | Get second character                   |
|       | ADD  | EBX,[TENS + EAX * 4]     | Add tens value                         |
|       | CALL | READ                     | Get third character                    |
|       | ADD  | EBX,EAX                  | EBX contains value of<br>binary number |
| READ: | BT   | INSTATUS,3               |                                        |
|       | JNC  | READ                     | Wait for new character                 |
|       | MOV  | AL,DATAIN                | Get new character                      |
|       | AND  | AL,0FH                   | Convert to equivalent<br>binary value  |
|       | RET  |                          |                                        |

3.66. (a) The subroutines convert 3 decimal digits to a binary value.

|           |      |                          |                                                |
|-----------|------|--------------------------|------------------------------------------------|
| GETCHARS: | PUSH | ECX                      | Save registers.                                |
|           | PUSH | EBX                      |                                                |
|           | PUSH | EAX                      |                                                |
|           | MOV  | ECX,3                    | Use ECX as character counter.                  |
|           | MOV  | EBX,[ESP + 20]           | Load character buffer address into EBX.        |
| READ:     | BT   | INSTATUS,3               |                                                |
|           | JNC  | READ                     |                                                |
|           | MOV  | BYTE PTR [EBX],DATAIN    | Get and store character.                       |
|           | INC  | EBX                      | Increment buffer pointer.                      |
|           | LOOP | READ                     | Repeat until all characters received.          |
|           | MOV  | EAX,[ESP + 16]           | Pointer to result.                             |
|           | CALL | CONVERT                  |                                                |
|           | POP  | EAX                      | Restore registers.                             |
|           | POP  | EBX                      |                                                |
|           | POP  | ECX                      |                                                |
|           | RET  |                          |                                                |
| CONVERT:  | PUSH | ECX                      | Save registers.                                |
|           | PUSH | EDX                      |                                                |
|           | DEC  | EBX                      | Load low-order digit numerical value into EDX. |
|           | MOV  | DL,[EBX]                 |                                                |
|           | AND  | DL,0FH                   |                                                |
|           | DEC  | EBX                      | Load and add tens digit value into EDX.        |
|           | MOV  | CL,[EBX]                 |                                                |
|           | AND  | CL,0FH                   |                                                |
|           | ADD  | EDX,[TENS + ECX * 4]     |                                                |
|           | DEC  | EBX                      | Load and add hundreds digit value into EDX.    |
|           | MOV  | CL,[EBX]                 |                                                |
|           | AND  | CL,0FH                   |                                                |
|           | ADD  | EDX,[HUNDREDS + ECX * 4] |                                                |
|           | MOV  | [EAX],EDX                | Store result.                                  |
|           | POP  | EDX                      | Restore registers.                             |
|           | POP  | ECX                      |                                                |
|           | RET  |                          |                                                |

(b) The contents of the top of the stack after the call to the CONVERT subroutine are:

|                            |
|----------------------------|
| ...                        |
| Return address to GETCHARS |
| [EAX]                      |
| [EBX]                      |
| [ECX]                      |
| Return address to Main     |
| Result address             |
| Buffer address             |
| ORIGINAL TOS               |
| ...                        |

3.67. Assume that the subroutine can change the contents of any registers used to pass parameters. Let  $\text{Stride} = 4m$ , which is the distance in bytes between successive doubleword elements in a given column.

|       |     |                     |                            |
|-------|-----|---------------------|----------------------------|
|       | SHL | EBX,2               | Set Stride in EBX.         |
|       | SUB | EDI,ESI             | Set EDI to $y - x$ .       |
|       | SHL | ESI,2               | Set EDX to                 |
|       | ADD | EDX,ESI             | address $A(0,x)$ .         |
| LOOP: | MOV | ESI,[EDX]           | Add $A(i,x)$ to $A(i,y)$ . |
|       | ADD | [EDX + EDI * 4],ESI |                            |
|       | ADD | EDX,EBX             | Move to next row.          |
|       | DEC | EAX                 | Repeat loop until all      |
|       | JG  | LOOP                | entries have been added.   |
|       | RET |                     | Return.                    |

3.68. Program trace:

| TIME      | EDI | ECX | DL  | LIST | LIST | LIST | LIST | LIST |
|-----------|-----|-----|-----|------|------|------|------|------|
|           |     |     |     |      | +1   | +2   | +3   | +4   |
| After 1st | 3   | -1  | 120 | 106  | 13   | 67   | 45   | 120  |
| After 2nd | 2   | -1  | 106 | 67   | 13   | 45   | 106  | 120  |
| After 3rd | 1   | -1  | 67  | 45   | 13   | 67   | 106  | 120  |
| After 4th | 0   | -1  | 45  | 13   | 45   | 67   | 106  | 120  |



3.69. Assume that the calling program passes the address `LIST - 4` to the subroutine in register `EAX`.

**Subroutine SORT**

|        |                                                                                                |                                                                                                                                                                                                                                                               |
|--------|------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SORT:  | PUSH EDI<br>PUSH ECX<br>PUSH EDX<br>MOV EDI,[EAX]<br>DEC EDI<br>ADD EAX,4                      | Save registers.<br><br><br>Initialize outer loop index register $EDI$ to $j = n - 1$ .<br>Set <code>EAX</code> to contain <code>LIST</code> .                                                                                                                 |
| OUTER: | MOV ECX,EDI<br>DEC ECX<br>MOV EDX,[EAX + EDI * 4]                                              | Initialize inner loop index register to $k = j - 1$ .<br>Load <code>LIST(j)</code> into <code>EDX</code> .                                                                                                                                                    |
| INNER: | CMP [EAX + ECX * 4],EDX<br>JLE NEXT<br><br>XCHG [EAX + ECX * 4],EDX<br>MOV [EAX + EDI * 4],EDX | Compare <code>LIST(k)</code> to <code>LIST(j)</code> .<br>If $LIST(k) \leq LIST(j)$ ,<br>go to next $k$ index entry;<br>Otherwise, interchange <code>LIST(k)</code><br>and <code>LIST(j)</code> , leaving<br>(new) <code>LIST(j)</code> in <code>EDX</code> . |
| NEXT:  | DEC ECX<br>JGE INNER<br>DEC EDI<br>JG OUTER<br>POP EDX<br>POP ECX<br>POP EDI<br>RET            | Decrement inner loop index $k$ .<br>Repeat or terminate inner loop.<br>Decrement outer loop index $j$ .<br>Repeat or terminate outer loop.<br>Restore registers.                                                                                              |

- 3.70. Use register ESI to keep track of the index position of the largest element in the inner loop, and use register EDX (DL) to record its value. Register EBX (BL) is used to hold sublist values to be compared to the current largest value.

```

                LEA    EAX,LIST
                MOV    EDI,N
                DEC    EDI
OUTER:          MOV    ECX,EDI
                DEC    ECX
                MOV    ESI,EDI      Initial index of largest.
                MOV    DL,[EAX + EDI] Initial value of largest.
INNER:          MOV    BL,[EAX + ECX] Get LIST(k) element.
                CMP    BL,DL        Compare to current largest.
                JLE    NEXT          If not larger, check next;
                MOV    DL,BL        Otherwise, update largest
                MOV    ESI,ECX      and update its index.
NEXT:           DEC    ECX          Repeat or terminate
                JGE    INNER        inner loop.
                XCHG   [EAX + EDI],DL Interchange LIST(j)
                MOV    [EAX + ESI],DL with LIST([ESI]).
                DEC    EDI          Repeat or terminate
                JG     OUTER        outer loop.

```

The potential advantage is that the inner loop should execute faster.

- 3.71. Assume that register ESI points to the first record, and use registers EAX, EBX, and ECX, to accumulate the three sums.

```

                MOV    EAX,0
                MOV    EBX,0
                MOV    ECX,0
LOOP:           ADD    EAX,[ESI + 8] Accumulate scores for test 1.
                ADD    EBX,[ESI + 12] Accumulate scores for test 2.
                ADD    ECX,[ESI + 16] Accumulate scores for test 3.
                MOV    ESI,[ESI + 4] Get link.
                CMP    ESI,0        Check if done.
                JNE    LOOP
                MOV    SUM1,EAX     Store sums.
                MOV    SUM2,EBX
                MOV    SUM3,ECX

```

- 3.72. If the ID of the new record matches the ID of the Head record of the current list, the new record will be inserted as the new Head. If the ID of the new record matches the ID of a later record in the current list, the new record will be inserted immediately after that record, including the case where the matching record is the Tail record. In this latter case, the new record becomes the new Tail record.

Modify Figure 3.51 as follows:

- Add the following instruction as the first instruction of the subroutine:

```

INSERTION:  MOV    EDX, 0                Anticipate successful
  insertion of the new
  record.
  MOV    RNEWID,[RNEWREC] (Existing instruction.)

```

- After the second CMP instruction, insert the following three instructions:

```

  JNE    CONTINUE1  Three new instructions.
  MOV    EDX,RHEAD
  RET
CONTINUE1:  JG     SEARCH                (Existing instruction.)

```

- After the fourth CMP instruction, insert the following three instructions:

```

  JNE    CONTINUE2  Three new instructions.
  MOV    EDX,RNEXT
  RET
CONTINUE2:  JL     INSERT                (Existing instruction.)

```

- 3.73. If the list is empty, the result is unpredictable because the first instruction will compare the ID of the new record to the contents of memory location zero. If the list is not empty, the following happens. If the contents of RIDNUM are less than the ID number of the Head record, the Head record will be deleted. Otherwise, the routine loops until register RCURRENT points to the Tail record. Then RNEXT gets loaded with zero by the instruction at LOOPSTART, and the result is unpredictable.

Replace Figure 3.52 with the following code:

|            |     |                      |                         |
|------------|-----|----------------------|-------------------------|
| DELETION:  | CMP | RHEAD, 0             | If the list is empty,   |
|            | JNE | CHECKHEAD            | return with RIDNUM      |
|            | RET |                      | unchanged.              |
| CHECKHEAD: | CMP | RIDNUM,[RHEAD]       | Check if Head record    |
|            | JNE | CONTINUE1            | is to be deleted and    |
|            | MOV | RHEAD,[RHEAD + 4]    | perform deletion if it  |
|            | MOV | RIDNUM,0             | is, returning with zero |
|            | RET |                      | in RIDNUM.              |
| CONTINUE1: | MOV | RCURRENT,RHEAD       | Otherwise, continue     |
|            |     |                      | searching.              |
| LOOPSTART: | MOV | RNEXT,[RCURRENT + 4] |                         |
|            | CMP | RNEXT,0              | If all records checked, |
|            | JNE | CHECKNEXT            | return with IDNUM       |
|            | RET |                      | unchanged.              |
| CHECKNEXT: | CMP | RIDNUM,[RNEXT]       | Check if next record is |
|            | JNE | CONTINUE2            | to be deleted and       |
|            | MOV | RTEMP,[RNEXT + 4]    | perform deletion if     |
|            | MOV | [RCURRENT + 4],RTEMP | it is, returning with   |
|            | MOV | RIDNUM,0             | zero in RIDNUM.         |
|            | RET |                      |                         |
| CONTINUE2: | MOV | RCURRENT,RNEXT       | Otherwise, continue     |
|            | JMP | LOOPSTART            | the search.             |

## Chapter 4 – Input/Output Organization

- 4.1. After reading the input data, it is necessary to clear the input status flag before the program begins a new read operation. Otherwise, the same input data would be read a second time.
- 4.2. The ASCII code for the numbers 0 to 9 can be obtained by adding \$30 to the number. The values 10 to 15 are represented by the letters A to F, whose ASCII codes can be obtained by adding \$37 to the corresponding binary number.

Assume the output status bit is  $b_4$  in register Status, and the output data register is Output.

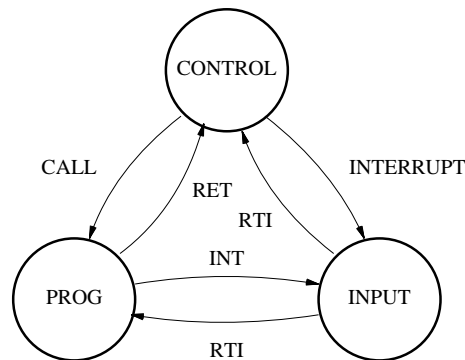
|         |                 |           |                                       |
|---------|-----------------|-----------|---------------------------------------|
| Next    | Move            | #10,R0    | Use R0 as counter                     |
|         | Move            | #LOC,R1   | Use R1 as pointer                     |
|         | Move            | (R1),R2   | Get next byte                         |
|         | Move            | R2,R3     |                                       |
|         | Shift-right     | #4,R3     | Prepare bits $b_7-b_4$                |
|         | Call            | Convert   |                                       |
|         | Move            | R2,R3     | Prepare bits $b_3-b_0$                |
|         | Call            | Convert   |                                       |
|         | Move            | \$20,R3   | Print space                           |
|         | Call            | Print     |                                       |
|         | Increment       | R1        |                                       |
|         | Decrement       | R0        |                                       |
|         | Branch>0        | Next      | Repeat if more bytes left             |
|         | End             |           |                                       |
| Convert | And             | #0F,R3    | Keep only low-order 4 bits            |
|         | Compare         | #9,R3     |                                       |
|         | Branch $\geq$ 0 | Letters   | Branch if $[R3] \geq 9$               |
|         | Or              | #\$30,R3  | Convert to ASCII, for values 0 to 9   |
|         | Branch          | Print     |                                       |
| Letters | Add             | #\$37,R3  | Convert to ASCII, for values 10 to 15 |
| Print   | BitTest         | #4,Status | Test output status bit                |
|         | Branch=0        | Print     | Loop back if equal to 0               |
|         | Move            | R3,Output | Send character to output register     |
|         | Return          |           |                                       |

4.3. 7CA4, 7DA4, 7EA4, 7FA4.

- 4.4. A subroutine is called by a program instruction to perform a function needed by the calling program. An interrupt-service routine is initiated by an event such as an input operation or a hardware error. The function it performs may not be at

all related to the program being executed at the time of interruption. Hence, it must not affect any of the data or status information relating to that program.

- 4.5. If execution of the interrupted instruction is to be completed after return from interrupt, a large amount of information needs to be saved. This includes the contents of any temporary registers, intermediate results, etc. An alternative is to abort the interrupted instruction and start its execution from the beginning after return from interrupt. In this case, the results of an instruction must not be stored in registers or memory locations until it is guaranteed that execution of the instruction will be completed without interruption.
- 4.6. (a) Interrupts should be enabled, except when C is being serviced. The nesting rules can be enforced by manipulating the interrupt-enable flags in the interfaces of A and B.  
 (b) A and B should be connected to  $\text{INTR}_2$ , and C to  $\text{INTR}_1$ . When an interrupt request is received from either A or B, interrupts from the other device will be automatically disabled until the request has been serviced. However, interrupt requests from C will always be accepted.
- 4.7. Interrupts are disabled before the interrupt-service routine is entered. Once device  $i$  turns off its interrupt request, interrupts may be safely enabled in the processor. If the interface circuit of device  $i$  turns off its interrupt request when it receives the interrupt acknowledge signal, interrupts may be enabled at the beginning of the interrupt-service routine of device  $i$ . Otherwise, interrupts may be enabled only after the instruction that causes device  $i$  to turn off its interrupt request has been executed.
- 4.8. Yes, because other devices may keep the interrupt request line asserted.
- 4.9. The control program includes an interrupt-service routine, INPUT, which reads the input characters. Transfer of control among various programs takes place as shown in the diagram below.



A number of status variables are required to coordinate the functions of PROG and INPUT, as follows.

**BLK-FULL:** A binary variable, indicating whether a block is full and ready for processing.

**IN-COUNT:** Number of characters read.

**IN-POINTER:** Points at the location where the next input character is to be stored.

**PROG-BLK:** Points at the location of the block to be processed by PROG.

Two memory buffers are needed, each capable of storing a block of data. Let BLK(0) and BLK(1) be the addresses of the two memory buffers. The structure of CONTROL and INPUT can be described as follows.

```
CONTROL  BLK-FULL := false
          IN-POINTER := BLK(0)
          IN-COUNT := 0
          Enable interrupts
          i := 0
          Loop
              Wait for BLK-FULL
              If not last block then {
                  BLK-FULL := false      Prepare to read the next block
                  IN-POINTER := BLK(1 - i)
                  IN-COUNT := 0
                  Enable interrupts }
              PROG-BLK := BLK(i)          Process the block just read
              Call PROG
              If last block then exit
              i := 1 - i
          End Loop
```

*Interrupt-service routine*

```
INPUT:    Store input character and increment IN-COUNT and IN-POINTER
          If IN-COUNT = N Then {
              disable interrupts from device
              BLK-FULL := true }
          Return from interrupt
```

4.10. *Correction: In the last paragraph, change “equivalent value” to “equivalent condition”.*

Assume that the interface registers for each video terminal are the same as in Figure 4.3. A list of device addresses is stored in the memory, starting at DEVICES, where the address given in the list, DEVADRS, is that of DATAIN. The pointers to data areas, PNTR<sub>n</sub>, are also stored in a list, starting at PNTRS.

Note that depending on the processor, several instructions may be needed to perform the function of one of the instructions used below.

|              |                 |                |                                    |
|--------------|-----------------|----------------|------------------------------------|
| POLL<br>LOOP | Move            | #20,R1         | Use R1 as device counter, $i$      |
|              | Move            | DEVICES(R1),R2 | Get address of device $i$          |
|              | BitTest         | #0,2(R2)       | Test input status of a device      |
|              | Branch $\neq$ 0 | NXTDV          | Skip read operation if not ready   |
|              | Move            | PNTRS(R1),R3   | Get pointer to data for device $i$ |
|              | MoveByte        | (R2),(R3)+     | Get and store input character      |
|              | Move            | R3,PNTRS(R1)   | Update pointer in memory           |
| NXTDV        | Decrement       | R1             |                                    |
|              | Branch $>$ 0    | LOOP           |                                    |
|              | Return          |                |                                    |

INTERRUPT Same as POLL, except that it returns once a character is read. If several devices are ready at the same time, the routine will be entered several times in succession.

In case  $a$ , POLL must be executed at least 100 times per second. Thus  $T_{max} = 10$  ms.

The equivalent condition for case  $b$  can be obtained by considering the case when all 20 terminals become ready at the same time. The time required for interrupt servicing must be less than the inter-character delay. That is,  $20 \times 200 \times 10^{-9} < 1/c$ , or  $c < 250,000$  char/s.

The time spent servicing the terminals in each second is given by:

$$\text{Case } a: \text{Time} = 100 \times 800 \times 10^{-9} \text{ ns} = 80 \mu\text{s}$$

$$\text{Case } b: \text{Time} = 20 \times r \times 200 \times 10^{-9} \times 100 = 400r \text{ ns}$$

Case  $b$  is a better strategy for  $r < 0.2$ .

The reader may repeat this problem using a slightly more complete model in which the polling time,  $P$ , for case  $a$  is a function of the number of terminals. For example, assume that  $P$  increases by  $0.5 \mu\text{s}$  for each terminal that is ready, that is,  $P = 20 + 20r \times 0.5$ .

- 4.11. (a) Read the interrupt vector number from the device (1 transfer).  
Save PC and SR (3 transfers on a 16-bit bus).  
Read the interrupt vector (2 transfers) and load it in the PC.
- (b) The 68000 instruction requiring the maximum number of memory transfers is:

MOVEM.L D0-D7/A0-A7,LOC.L

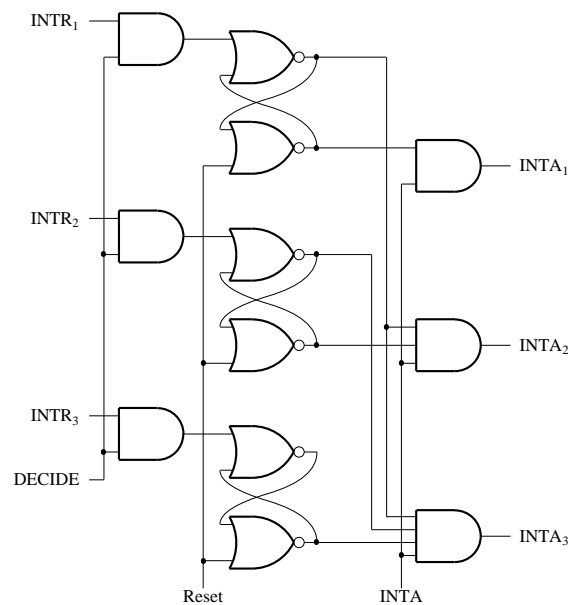
where LOC.L is a 32-bit absolute address. Four memory transfers are needed to read the instruction, followed by 2 transfers for each register, for a total of 36.

- (c) 36 for completion of current instruction plus 6 for interrupt handling, for a total of 42.

- 4.12. (a)  $\text{INTA1} = \text{INTR1}$   
 $\text{INTA2} = \text{INTR2} \cdot \overline{\text{INTR1}}$   
 $\text{INTA3} = \text{INTR3} \cdot \overline{\text{INTR1}} \cdot \overline{\text{INTR2}}$

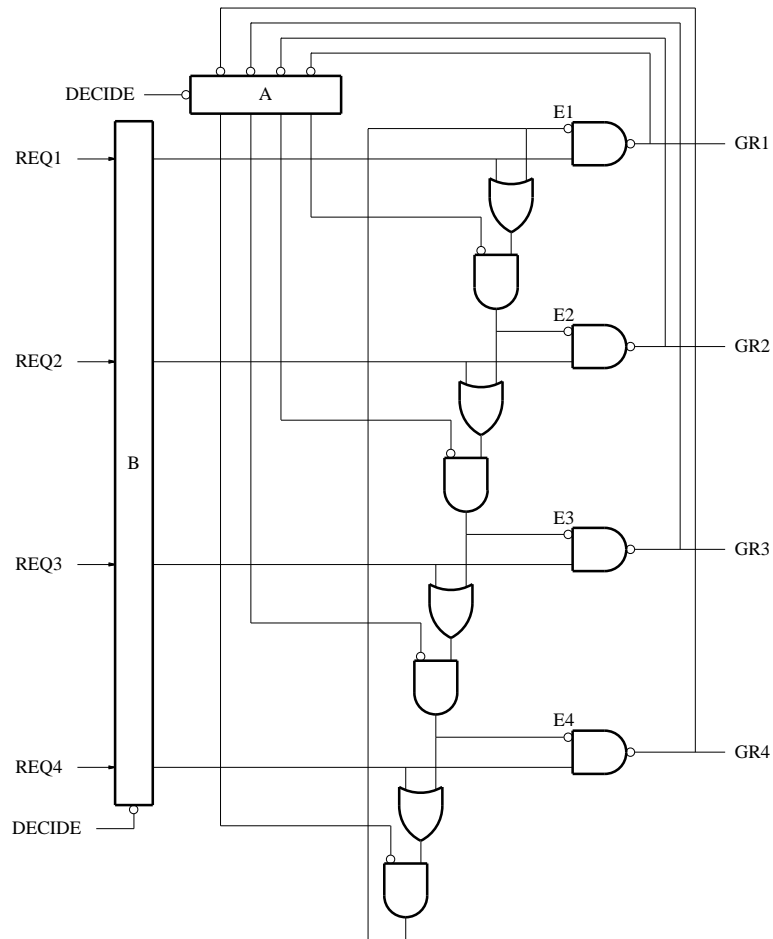


- (b) See logic equations in part *a*.
- (c) Yes.
- (d) In the circuit below, DECIDE is used to lock interrupt requests. The processor should set the interrupt acknowledge signal, INTA, after DECIDE returns to zero. This will cause the highest priority request to be acknowledged. Note that latches are placed at the inputs of the priority circuit. They could be placed at the outputs, but the circuit would be less reliable when interrupts change at about the same time as arbitration is taking place (races may occur).



- 4.13. In the circuit given below, register A records which device was given a grant most recently. Only one of its outputs is equal to 1 at any given time, identifying the highest-priority line. The falling edge of DECIDE records the results of the current arbitration cycle in A and at the same time records new requests in register B. This prevents requests that arrive later from changing the grant.

The circuit requires careful initialization, because one and only one output of register A must be equal to 1. This output determines the highest-priority line during a given arbitration cycle. For example, if the LSB of A is equal to 1, point E2 will be equal to 0, giving REQ2 the highest priority.



4.14. The truth table for a priority encoder is given below.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | IPL <sub>2</sub> | IPL <sub>1</sub> | IPL <sub>0</sub> |
|---|---|---|---|---|---|---|------------------|------------------|------------------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0                | 0                | 0                |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0                | 0                | 1                |
| x | 1 | 0 | 0 | 0 | 0 | 0 | 0                | 1                | 0                |
| x | x | 1 | 0 | 0 | 0 | 0 | 0                | 1                | 1                |
| x | x | x | 1 | 0 | 0 | 0 | 1                | 0                | 0                |
| x | x | x | x | 1 | 0 | 0 | 1                | 0                | 1                |
| x | x | x | x | x | 1 | 0 | 1                | 1                | 0                |
| x | x | x | x | x | x | 1 | 1                | 1                | 1                |

A possible implementation for this priority circuit is as follows:

$$IPL_2 = q_4 + q_5 + q_6 + q_7$$

$$IPL_1 = q_6 + q_7 + \overline{IPL_2}(q_2 + q_3)$$

$$IPL_0 = q_7 + q_5 \cdot q_6 + \overline{IPL_2}(q_3 + q_1 \cdot q_2)$$

- 4.15. Assume that the interface registers are the same as in Figure 4.3 and that the characters to be printed are stored in the memory.

```

* Program A (MAIN) points to the character string and calls DSPLY twice
MAIN      MOVE.L    #ISR,VECTOR    Initialize interrupt vector
          ORI.B     #$80,STATUS    Enable interrupts from device
          MOVE      #$2300,SR      Set interrupt mask to 3
          MOVEA.L   #CHARS,A0     Set pointer to character list
          BSR       DSPLY
          MOVEA.L   #CHARS,A0
          BSR       DSPLY
          END       MAIN

* Subroutine DSPLY prints the character string pointed to by A0
* The last character in the string must be the NULL character
DSPLY     ...
          RTS

* Program B, the interrupt-service routine, points at the number string and calls DSPLY
ISR       MOVEM.L   A0,-(A7)       Save registers used
          MOVE.L    NEWLINE,A0    Start a new line
          BSR       DSPLY
          MOVEA.L   #NMBRS,A0     Point to the number string
          BSR       DSPLY
          MOVEM.L   (A7)+,A0       Restore registers
          RTE

* Characters and numbers to be displayed
CHARS     CC        /AB ... Z/
NEWLINE   CB        $0D, $0A, 0    Codes for CR, LF and Null
NMBRS     CB        $0D, $0A
          CC        /01 ... 901 ... 901 ... 9/
          CB        $0D, $0A, 0

```

When ISR is entered, the interrupt mask in SR is automatically set to 4 by the hardware. To allow interrupt nesting, the mask must be set to 3 at the beginning of ISR.

- 4.16. Modify subroutine DSPLY in Problem 4.15 to keep count of the number of characters printed in register D1. Before ISR returns, it should call RESTORE, which sends a number of space characters (ASCII code 20<sub>16</sub>) equal to the count in D1.

|         |        |               |                             |
|---------|--------|---------------|-----------------------------|
| DSPLY   | ...    |               |                             |
|         | MOVE   | #\$2400,SR    | Disable keyboard interrupts |
|         | MOVEB  | D0,DATAOUT    | Print character             |
|         | ADDQ   | #1,D1         |                             |
|         | MOVE   | #\$2300,SR    | Enable keyboard interrupts  |
|         | ...    |               |                             |
| RESTORE | MOVE.L | D1,D2         |                             |
|         | BR     | TEST          |                             |
| LOOP    | BTST   | #1,STATUS     |                             |
|         | BEQ    | LOOP          |                             |
|         | MOVEB  | #\$20,DATAOUT |                             |
| TEST    | DBRA   | D2,LOOP       |                             |
|         | RTS    |               |                             |

Note that interrupts are disabled in DSPLY before printing a character to ensure that no further interrupts are accepted until the count is updated.

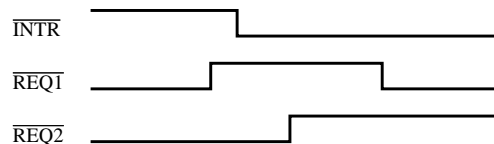
- 4.17. The debugger can use the trace interrupt to execute the saved instruction then regain control. The debugger puts the saved instruction at the correct address, enables trace interrupts and returns. The instruction will be executed. Then, a second interruption will occur, and the debugger begins execution again. The debugger can now remove the program instruction, reinstall the breakpoint, disable trace interrupts, then return to resume program execution.
- 4.18. (a) The return address, which is in register R14\_svc, is PC+4, where PC is the address of the SWI instruction.

|     |                  |                       |
|-----|------------------|-----------------------|
| LDR | R2,[R14,#-4]     | Get SWI instruction   |
| BIC | R2,R2,#&FFFFFF00 | Clear high-order bits |

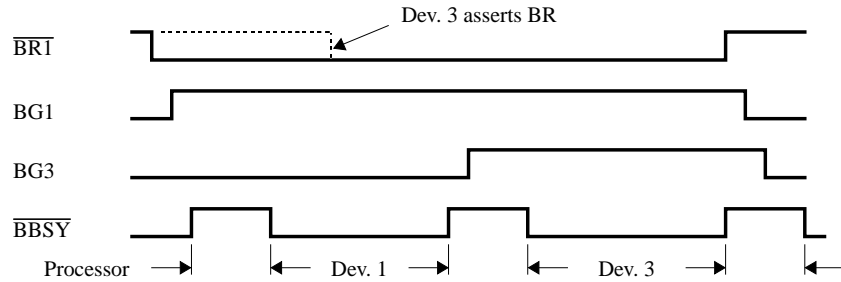
- (b) Assume that the low-order 8 bits in SWI have the values 1, 2, 3, ... to request services number 1, 2, 3, etc. Use register R3 to point to a table of addresses of the corresponding routines, at addresses [R3]+4, [R3]+8, respectively.

|     |                    |                                  |
|-----|--------------------|----------------------------------|
| ADR | R3,EntryTable      | Get the table's address          |
| LDR | R15,[R3,R2,LSL #2] | Load starting address of routine |

- 4.19. Each device pulls the line down (closes a switch to ground) when it is not ready. It opens the switch when it is ready. Thus, the line will be high when all devices are ready.
- 4.20. The request from one device may be masked by the other, because the processor may see only one edge.

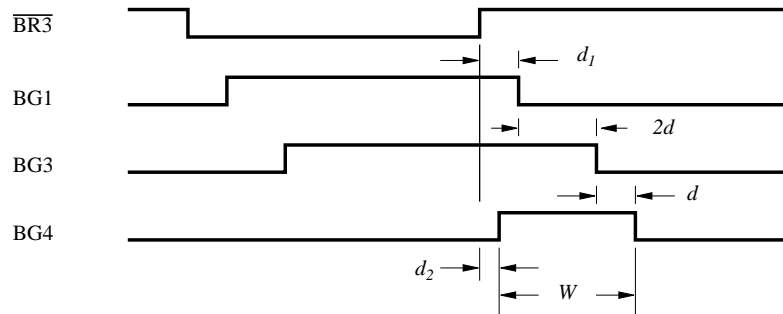


- 4.21. Assume that when BR becomes active, the processor asserts BG1 and keeps it asserted until BR is negated.



- 4.22. (a) Device 2 requests the bus and receives a grant. Before it releases the bus, device 1 also asserts BR. When device 2 is finished nothing will happen. BR and BG1 remain active, but since device 1 does not see a transition on BG1 it cannot become the bus master.  
(b) No device may assert BR if its BG input is active.

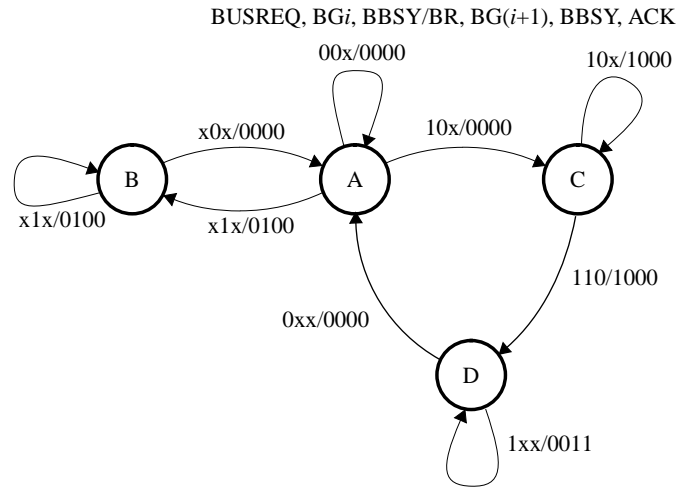
- 4.23. For better clarity, change BR to  $\overline{\text{BR}}$  and use an inverter with delay  $d_1$  to generate BG1.



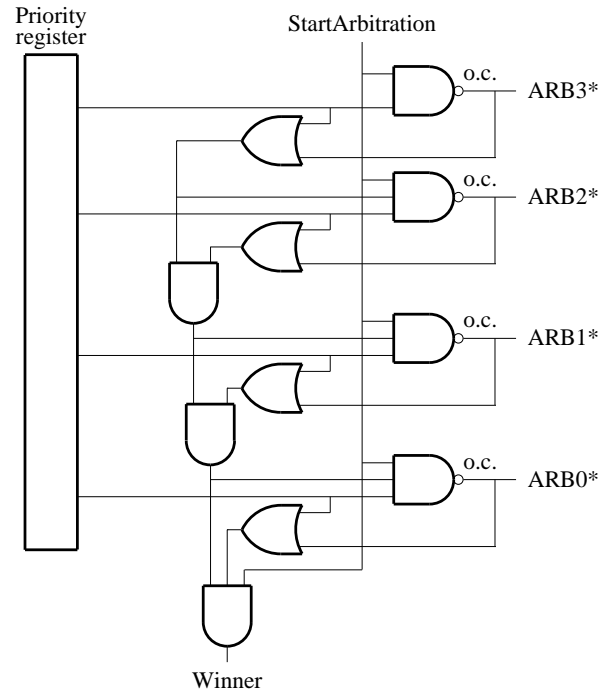
Assuming device 3 asserts BG4 shortly after it drops the bus request (delay  $d_2$ ), a spurious pulse of width  $W = d_1 + 3d - d_2$  will appear on BG4.

- 4.24. Refer to the timing diagram in Problem 4.23. Assume that both BR1 and BR5 are activated during the delay period  $d_2$ . Input BG1 will become active and at the same time the pulse on BG4 will travel to BG5. Thus, both devices will receive a bus grant at the same time.

- 4.25. A state machine for the required circuit is given in the figure below. An output called ACK has been added, indicating when the device may use the bus. Note that the restriction in Solution 4.22b above is observed (state B).



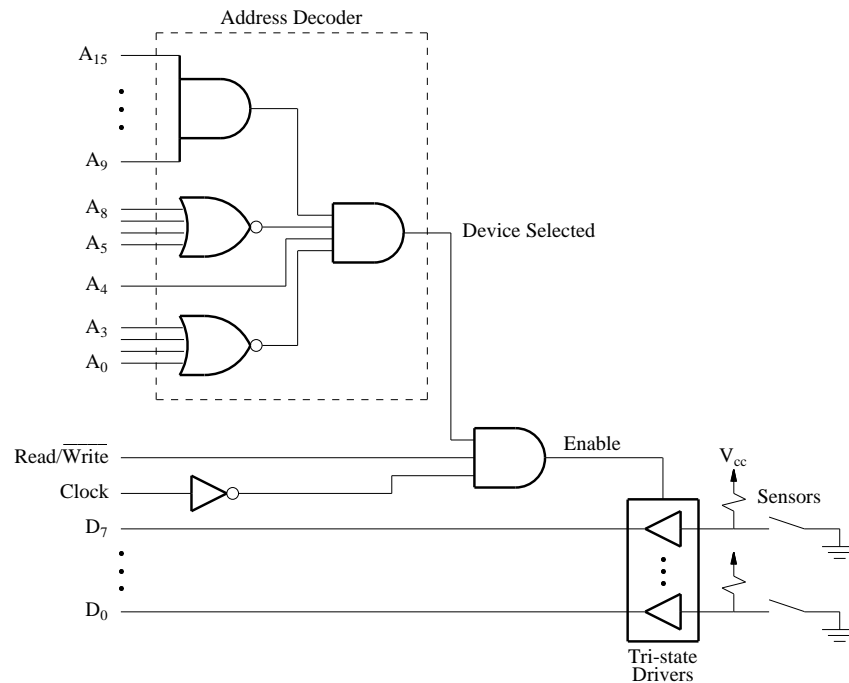
- 4.26. The priority register in the circuit below contains 1111 for the highest priority device and 0000 for the lowest.



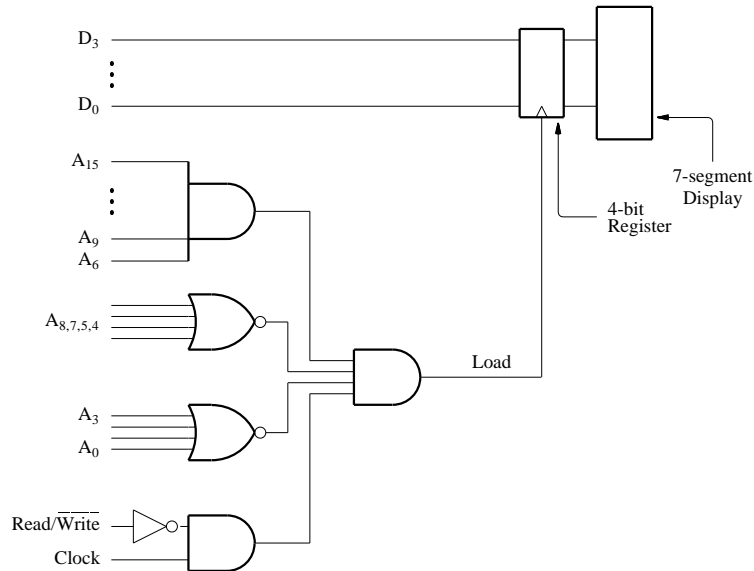
4.27. A larger distance means longer delay for the signals traveling between the processor and the input device. Primarily, this means that  $t_2 - t_1$ ,  $t_3 - t_2$  and  $t_5 - t_3$  will increase. Since longer distances may also mean larger skew, the intervals  $t_1 - t_0$  and  $t_4 - t_3$  may have to be increased to cover worst-case differences in propagation delay.

In the case of Figure 4.24, the clock period must be increased to accommodate the maximum propagation delay.

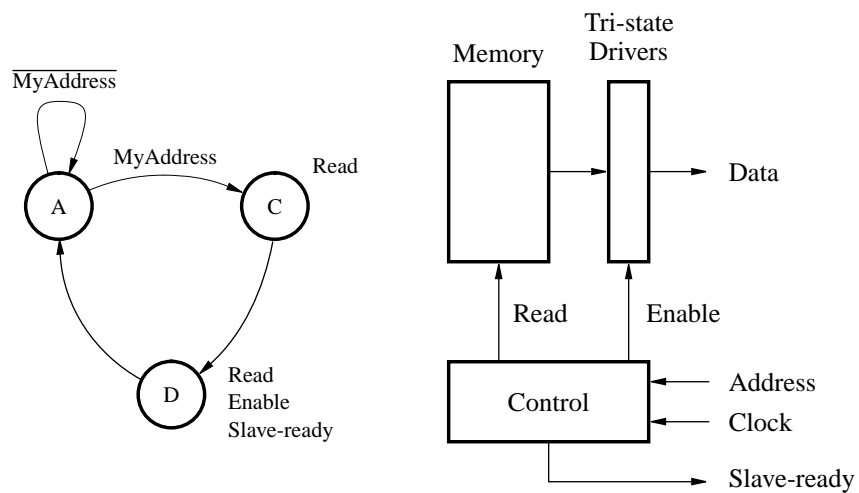
4.28. A possible circuit is given below.



- 4.29. Assume that the display has the bus address FE40. The circuit below sets the Load signal to 0 during the second half of the write cycle. The rising edge at the end of the clock period will load the data into the display register.



- 4.30. Generate  $SIN_{write}$  in the same way as Load in Problem P4.29. This signal should load the data on D6 into an Interrupt-Enable flip-flop, IntEn. The interrupt request can now be generated as  $\overline{INTR} = \overline{SIN} \cdot IntEn$ .
- 4.31. Hardware organization and a state diagram for the memory interface circuit are given below.





4.32. (a) Once the memory receives the address and data, the bus is no longer needed. Operations involving other devices can proceed.

(b) The bus protocol may be designed such that no response is needed for write operations, provided that arrival of the address and data in the first clock cycle is guaranteed. The main precaution that must be taken is that the memory interface cannot respond to other requests until it has completed the write operation. Thus, a subsequent read or write operation may encounter additional delay.

Note that without a response signal the processor is not informed if the memory does not receive the data for any reason. Also, we have assumed a simple uni-processor environment. For a discussion of the constraints in parallel-processing systems, see Chapter 12.

4.33. In the case of Figure 4.24, the lack of response will not be detected and processing will continue, leading to erroneous results. For this reason, a response signal from the device should be provided, even though it is not essential for bus operation. The schemes of both Figures 4.25 and 4.26 provide a response signal, Slave-ready. No response would cause the bus to hang up. Thus, after some time-out period the processor should abort the transaction and begin executing an appropriate bus error exception routine.

4.34. The device may contain a buffer to hold the address value if it requires additional time to decode it or to access the requested data. In this case, the address may be removed from the bus after the first cycle.

4.35. Minimum clock period =  $4+5+6+10+3 = 28$  ns  
Maximum clock speed = 35.7 MHz

These calculations assume no clock skew between the sender and the receiver.

4.36.  $t_1 - t_0 \geq \text{bus skew} = 4$  ns  
 $t_2 - t_1 = \text{propagation delay} + \text{address decoding} + \text{access time}$   
 $= 1 \text{ to } 5 + 6 + 5 \text{ to } 10 = 12 \text{ to } 21$  ns  
 $t_3 - t_2 = \text{propagation delay} + \text{skew} + \text{setup time}$   
 $= 1 \text{ to } 5 + 4 + 3 = 8 \text{ to } 12$  ns  
 $t_5 - t_3 = \text{propagation delay} = 1 \text{ to } 5$  ns  
Minimum cycle =  $4 + 12 + 8 + 1 = 25$  ns  
Maximum cycle =  $4 + 21 + 12 + 5 = 42$  ns

# Chapter 5 – The Memory System

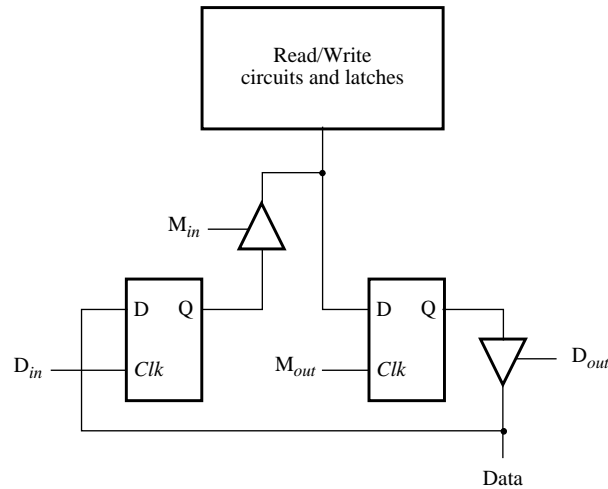
5.1. The block diagram is essentially the same as in Figure 5.10, except that 16 rows (of four  $512 \times 8$  chips) are needed. Address lines  $A_{18-0}$  are connected to all chips. Address lines  $A_{22-19}$  are connected to a 4-bit decoder to select one of the 16 rows.

5.2. The minimum refresh rate is given by

$$\frac{50 \times 10^{-15} \times (4.5 - 3)}{9 \times 10^{-12}} = 8.33 \times 10^{-3} \text{ s}$$

Therefore, each row has to be refreshed every 8 ms.

5.3. Need control signals  $M_{in}$  and  $M_{out}$  to control storing of data into the memory cells and to gate the data read from the memory onto the bus, respectively. A possible circuit is



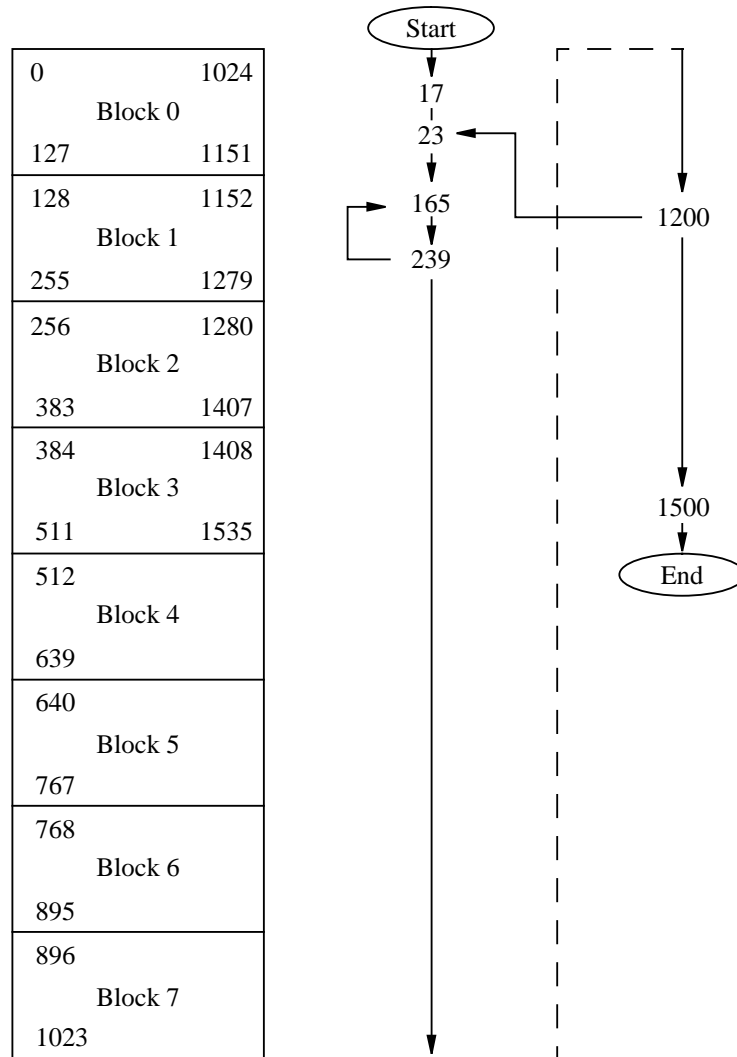
5.4. (a) It takes  $5 + 8 = 13$  clock cycles.

$$\text{Total time} = \frac{13}{(133 \times 10^6)} = 0.098 \times 10^{-6} \text{ s} = 98 \text{ ns}$$

$$\text{Latency} = \frac{5}{(133 \times 10^6)} = 0.038 \times 10^{-6} \text{ s} = 38 \text{ ns}$$

(b) It takes twice as long to transfer 64 bytes, because two independent 32-byte transfers have to be made. The latency is the same, i.e. 38 ns.

- 5.5. A faster processor chip will result in increased performance, but the amount of increase will not be directly proportional to the increase in processor speed, because the cache miss penalty will remain the same if the main memory speed is not improved.
- 5.6. (a) Main memory address length is 16 bits. TAG field is 6 bits. BLOCK field is 3 bits (8 blocks). WORD field is 7 bits (128 words per block).
- (b) The program words are mapped on the cache blocks as follows:



Hence, the sequence of reads from the main memory blocks into cache blocks is

Block : 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 0, 1, 0, 1, ..., 0, 1, 0, 1, 0, 1, 0, 1, 2, 3

Pass 1
Pass 2
Pass 9
Pass 10

As this sequence shows, both the beginning and the end of the outer loop use blocks 0 and 1 in the cache. They overwrite each other on each pass through the loop. Blocks 2 to 7 remain resident in the cache until the outer loop is completed.

The total time for reading the blocks from the main memory into the cache is therefore

$$(10 + 4 \times 9 + 2) \times 128 \times 10 \tau = 61,440 \tau$$

Executing the program out of the cache:

$$\text{Outer loop - inner loop} = [(1200 - 22) - (239 - 164)]10 \times 1\tau = 11,030 \tau$$

$$\text{Inner loop} = (239 - 164)200 \times 1\tau = 15,000 \tau$$

$$\text{End section of program} = 1500 - 1200 = 300 \times 1\tau$$

$$\text{Total execution time} = 87,770 \tau$$

5.7. In the first pass through the loop, the Add instruction is stored at address 4 in the cache, and its operand (A03C) at address 6. Then the operand is overwritten by the Decrement instruction. The BNE instruction is stored at address 0. In the second pass, the value 05D9 overwrites the BNE instruction, then BNE is read from the main memory and again stored in location 0. The contents of the cache, the number of words read from the main memory and from the cache, and the execution time for each pass are as shown below.

| After pass No. | Cache contents | MM accesses | Cache accesses | Time      |
|----------------|----------------|-------------|----------------|-----------|
| 1              | 005E BNE       | 4           | 0              | 40 $\tau$ |
|                |                |             |                |           |
|                | 005D Add       |             |                |           |
|                | 005D Dec       |             |                |           |
| 2              | 005E BNE       | 2           | 2              | 22 $\tau$ |
|                |                |             |                |           |
|                | 005D Add       |             |                |           |
|                | 005D Dec       |             |                |           |
| 3              | 005E BNE       | 1           | 3              | 13 $\tau$ |
|                | 00AA 10D7      |             |                |           |
|                | 005D Add       |             |                |           |
|                | 005D Dec       |             |                |           |
| Total          |                | 7           | 5              | 75 $\tau$ |

- 5.8. All three instructions are stored in the cache after the first pass, and they remain in place during subsequent passes. In this case, there is a total of 6 read operations from the main memory and 6 from the cache. Execution time is  $66\tau$ . Instructions and data are best stored in separate caches to avoid the data overwriting instructions, as in Problem 5.7.
- 5.9. (a) 4096 blocks of 128 words each require  $12 + 7 = 19$  bits for the main memory address.  
 (b) TAG field is 8 bits. SET field is 4 bits. WORD field is 7 bits.
- 5.10. (a) TAG field is 10 bits. SET field is 4 bits. WORD field is 6 bits.  
 (b) Words 0, 1, 2, ..., 4351 occupy blocks 0 to 67 in the main memory (MM). After blocks 0, 1, 2, ..., 63 have been read from MM into the cache on the first pass, the cache is full. Because of the fact that the replacement algorithm is LRU, MM blocks that occupy the first four sets of the 16 cache sets are always overwritten before they can be used on a successive pass. In particular, MM blocks 0, 16, 32, 48, and 64 continually displace each other in competing for the 4 block positions in cache set 0. The same thing occurs in cache set 1 (MM blocks 1, 17, 33, 49, 65), cache set 2 (MM blocks 2, 18, 34, 50, 66) and cache set 3 (MM blocks 3, 19, 35, 51, 67). MM blocks that occupy the last 12 sets (sets 4 through 15) are fetched once on the first pass and remain in the cache for the next 9 passes. On the first pass, all 68 blocks of the loop must be fetched from the MM. On each of the 9 successive passes, blocks in the last 12 sets of the cache ( $4 \times 12 = 48$ ) are found in the cache, and the remaining 20 ( $68 - 48$ ) blocks must be fetched from the MM.

$$\begin{aligned}
 \text{Improvement factor} &= \frac{\text{Time without cache}}{\text{Time with cache}} \\
 &= \frac{10 \times 68 \times 10\tau}{1 \times 68 \times 11\tau + 9(20 \times 11\tau + 48 \times 1\tau)} \\
 &= 2.15
 \end{aligned}$$

- 5.11. This replacement algorithm is actually better on this particular "large" loop example. After the cache has been filled by the main memory blocks 0, 1, ..., 63 on the first pass, block 64 replaces block 48 in set 0. On the second pass, block 48 replaces block 32 in set 0. On the third pass, block 32 replaces block 16, and on the fourth pass, block 16 replaces block 0. On the fourth pass, there are two replacements: 0 kicks out 64, and 64 kicks out 48. On the sixth, seventh, and eighth passes, there is only one replacement in set 0. On the ninth pass there are two replacements in set 0, and on the final pass there is one replacement. The situation is similar in sets 1, 2, and 3. Again, there is no contention in sets 4 through 15. In total, there are 11 replacements in set 0 in passes 2 through 10. The same is true in sets 1, 2, and 3. Therefore, the improvement factor is

$$\frac{10 \times 68 \times 10\tau}{1 \times 68 \times 11\tau + 4 \times 11 \times 11\tau + (9 \times 68 - 44) \times 1\tau} = 3.8$$

5.12. For the first loop, the contents of the cache are as indicated in Figures 5.20 through 5.22. For the second loop, they are as follows.

(a) Direct-mapped cache

| Block position | Contents of data cache after pass: |         |         |         |         |         |
|----------------|------------------------------------|---------|---------|---------|---------|---------|
|                | $j = 9$                            | $i = 1$ | $i = 3$ | $i = 5$ | $i = 7$ | $i = 9$ |
| 0              | A(0,8)                             | A(0,0)  | A(0,2)  | A(0,4)  | A(0,6)  | A(0,8)  |
| 1              |                                    |         |         |         |         |         |
| 2              |                                    |         |         |         |         |         |
| 3              |                                    |         |         |         |         |         |
| 4              | A(0,9)                             | A(0,1)  | A(0,3)  | A(0,5)  | A(0,7)  | A(0,9)  |
| 5              |                                    |         |         |         |         |         |
| 6              |                                    |         |         |         |         |         |
| 7              |                                    |         |         |         |         |         |

(b) Associative-mapped cache

| Block position | Contents of data cache after pass: |         |         |         |
|----------------|------------------------------------|---------|---------|---------|
|                | $j = 9$                            | $i = 0$ | $i = 5$ | $i = 9$ |
| 0              | A(0,8)                             | A(0,8)  | A(0,8)  | A(0,6)  |
| 1              | A(0,9)                             | A(0,9)  | A(0,9)  | A(0,7)  |
| 2              | A(0,2)                             | A(0,0)  | A(0,0)  | A(0,8)  |
| 3              | A(0,3)                             | A(0,3)  | A(0,1)  | A(0,9)  |
| 4              | A(0,4)                             | A(0,4)  | A(0,2)  | A(0,2)  |
| 5              | A(0,5)                             | A(0,5)  | A(0,3)  | A(0,3)  |
| 6              | A(0,6)                             | A(0,6)  | A(0,4)  | A(0,4)  |
| 7              | A(0,7)                             | A(0,7)  | A(0,5)  | A(0,5)  |

(c) Set-associative-mapped cache

|       |   | Contents of data cache after pass: |         |         |         |
|-------|---|------------------------------------|---------|---------|---------|
|       |   | $j = 9$                            | $i = 3$ | $i = 7$ | $i = 9$ |
| Set 0 | 0 | A(0,8)                             | A(0,2)  | A(0,6)  | A(0,6)  |
|       | 1 | A(0,9)                             | A(0,3)  | A(0,7)  | A(0,7)  |
|       | 2 | A(0,6)                             | A(0,0)  | A(0,4)  | A(0,8)  |
|       | 3 | A(0,7)                             | A(0,1)  | A(0,5)  | A(0,9)  |
| Set 1 | 0 |                                    |         |         |         |
|       | 1 |                                    |         |         |         |
|       | 2 |                                    |         |         |         |
|       | 3 |                                    |         |         |         |

In all 3 cases, all elements are overwritten before they are used in the second loop. This suggests that the LRU algorithm may not lead to good performance if used with arrays that do not fit into the cache. The performance can be improved by introducing some randomness in the replacement algorithm.

- 5.13. The two least-significant bits of an address,  $A_{1-0}$ , specify a byte within a 32-bit word. For a direct-mapped cache, bits  $A_{4-2}$  specify the block position. For a set-associative-mapped cache, bit  $A_2$  specifies the set.

(a) Direct-mapped cache

|                |   | Contents of data cache after: |        |        |        |
|----------------|---|-------------------------------|--------|--------|--------|
|                |   | Pass 1                        | Pass 2 | Pass 3 | Pass 4 |
| Block position | 0 | [200]                         | [200]  | [200]  | [200]  |
|                | 1 | [204]                         | [204]  | [204]  | [204]  |
|                | 2 | [208]                         | [208]  | [208]  | [208]  |
|                | 3 | [24C]                         | [24C]  | [24C]  | [24C]  |
|                | 4 | [2F0]                         | [2F0]  | [2F0]  | [2F0]  |
|                | 5 | [2F4]                         | [2F4]  | [2F4]  | [2F4]  |
|                | 6 | [218]                         | [218]  | [218]  | [218]  |
|                | 7 | [21C]                         | [21C]  | [21C]  | [21C]  |

$$\text{Hit rate} = 33/48 = 0.69$$

(b) Associative-mapped cache

| Block position | Contents of data cache after: |        |        |        |
|----------------|-------------------------------|--------|--------|--------|
|                | Pass 1                        | Pass 2 | Pass 3 | Pass 4 |
| 0              | [200]                         | [200]  | [200]  | [200]  |
| 1              | [204]                         | [204]  | [204]  | [204]  |
| 2              | [24C]                         | [21C]  | [218]  | [2F0]  |
| 3              | [20C]                         | [24C]  | [21C]  | [218]  |
| 4              | [2F4]                         | [2F4]  | [2F4]  | [2F4]  |
| 5              | [2F0]                         | [20C]  | [24C]  | [21C]  |
| 6              | [218]                         | [2F0]  | [20C]  | [24C]  |
| 7              | [21C]                         | [218]  | [2F0]  | [20C]  |

Hit rate =  $21/48 = 0.44$

(c) Set-associative-mapped cache

|       | Block position | Contents of data cache after: |        |        |        |
|-------|----------------|-------------------------------|--------|--------|--------|
|       |                | Pass 1                        | Pass 2 | Pass 3 | Pass 4 |
| Set 0 | 0              | [200]                         | [200]  | [200]  | [200]  |
|       | 1              | [208]                         | [208]  | [208]  | [208]  |
|       | 2              | [2F0]                         | [2F0]  | [2F0]  | [2F0]  |
|       | 3              | [218]                         | [218]  | [218]  | [218]  |
| Set 1 | 0              | [204]                         | [204]  | [204]  | [204]  |
|       | 1              | [24C]                         | [21C]  | [24C]  | [21C]  |
|       | 2              | [2F4]                         | [2F4]  | [2F4]  | [2F4]  |
|       | 3              | [21C]                         | [24C]  | [21C]  | [24C]  |

Hit rate =  $30/48 = 0.63$



- 5.14. The two least-significant bits of an address,  $A_{1-0}$ , specify a byte within a 32-bit word. For a direct-mapped cache, bits  $A_{4-3}$  specify the block position. For a set-associative-mapped cache, bit  $A_3$  specifies the set.

(a) Direct-mapped cache

| Block position | Contents of data cache after: |        |        |        |
|----------------|-------------------------------|--------|--------|--------|
|                | Pass 1                        | Pass 2 | Pass 3 | Pass 4 |
| 0              | [200]                         | [200]  | [200]  | [200]  |
|                | [204]                         | [204]  | [204]  | [204]  |
| 1              | [248]                         | [248]  | [248]  | [248]  |
|                | [24C]                         | [24C]  | [24C]  | [24C]  |
| 2              | [2F0]                         | [2F0]  | [2F0]  | [2F0]  |
|                | [2F4]                         | [2F4]  | [2F4]  | [2F4]  |
| 3              | [218]                         | [218]  | [218]  | [218]  |
|                | [21C]                         | [21C]  | [21C]  | [21C]  |

Hit rate =  $37/48 = 0.77$

(b) Associative-mapped cache

| Block position | Contents of data cache after: |        |        |        |
|----------------|-------------------------------|--------|--------|--------|
|                | Pass 1                        | Pass 2 | Pass 3 | Pass 4 |
| 0              | [200]                         | [200]  | [200]  | [200]  |
|                | [204]                         | [204]  | [204]  | [204]  |
| 1              | [248]                         | [218]  | [248]  | [218]  |
|                | [24C]                         | [21C]  | [24C]  | [21C]  |
| 2              | [2F0]                         | [2F0]  | [2F0]  | [2F0]  |
|                | [2F4]                         | [2F4]  | [2F4]  | [2F4]  |
| 3              | [218]                         | [248]  | [218]  | [248]  |
|                | [21C]                         | [24C]  | [21C]  | [24C]  |

Hit rate =  $34/48 = 0.71$

(c) Set-associative-mapped cache

|       |   | Contents of data cache after: |        |        |        |
|-------|---|-------------------------------|--------|--------|--------|
|       |   | Pass 1                        | Pass 2 | Pass 3 | Pass 4 |
| Set 0 | 0 | [200]                         | [200]  | [200]  | [200]  |
|       |   | [204]                         | [204]  | [204]  | [204]  |
|       | 1 | [2F0]                         | [2F0]  | [2F0]  | [2F0]  |
|       |   | [2F4]                         | [2F4]  | [2F4]  | [2F4]  |
| Set 1 | 0 | [248]                         | [218]  | [248]  | [218]  |
|       |   | [24C]                         | [21C]  | [24C]  | [21C]  |
|       | 1 | [218]                         | [248]  | [218]  | [248]  |
|       |   | [21C]                         | [24C]  | [21C]  | [24C]  |

Hit rate =  $34/48 = 0.71$

5.15. The block size (number of words in a block) of the cache should be at least as large as  $2^k$ , in order to take full advantage of the multiple module memory when transferring a block between the cache and the main memory. Power of 2 multiples of  $2^k$  work just as efficiently, and are natural because block size is  $2^k$  for  $k$  bits in the "word" field.

5.16. Larger size

- fewer misses if most of the data in the block are actually used
- wasteful if much of the data are not used before the cache block is ejected from the cache

Smaller size

- more misses

5.17. For 16-word blocks the value of  $M$  is  $1 + 8 + 3 \times 4 + 4 = 25$  cycles. Then

$$\frac{\text{Time without cache}}{\text{Time with cache}} = 4.04$$

In order to compare the 8-word and 16-word blocks, we can assume that two 8-word blocks must be brought into the cache for each 16-word block. Hence, the effective value of  $M$  is  $2 \times 17 = 34$ . Then

$$\frac{\text{Time without cache}}{\text{Time with cache}} = 3.3$$

Similarly, for 4-word blocks the effective value of  $M$  is  $4(1+8+4) = 52$  cycles.

Then

$$\frac{\text{Time without cache}}{\text{Time with cache}} = 2.42$$

Clearly, interleaving is more effective if larger cache blocks are used.

5.18. The hit rates are

$$\begin{aligned} h_1 = h_2 = h &= 0.95 \text{ for instructions} \\ &= 0.90 \text{ for data} \end{aligned}$$

The average access time is computed as

$$t_{ave} = hC_1 + (1-h)hC_2 + (1-h)^2M$$

(a) With interleaving  $M = 17$ . Then

$$\begin{aligned} t_{ave} &= 0.95 \times 1 + 0.05 \times 0.95 \times 10 + 0.0025 \times 17 + 0.3(0.9 \times 1 + 0.1 \times 0.9 \times 10 + 0.01 \times 17) \\ &= 2.0585 \text{ cycles} \end{aligned}$$

(b) Without interleaving  $M = 38$ . Then  $t_{ave} = 2.174$  cycles.

(c) Without interleaving the average access takes  $2.174/2.0585 = 1.056$  times longer.

5.19. Suppose that it takes one clock cycle to send the address to the L2 cache, one cycle to access each word in the block, and one cycle to transfer a word from the L2 cache to the L1 cache. This leads to  $C_2 = 6$  cycles.

(a) With interleaving  $M = 1 + 8 + 4 = 13$ . Then  $t_{ave} = 1.79$  cycles.

(b) Without interleaving  $M = 1 + 8 + 3 \times 4 + 1 = 22$ . Then  $t_{ave} = 1.86$  cycles.

(c) Without interleaving the average access takes  $1.86/1.79 = 1.039$  times longer.

5.20. The analogy is good with respect to:

- relative sizes of toolbox, truck and shop versus L1 cache, L2 cache and main memory
- relative access times
- relative frequency of use of tools in the 3 storage places versus the data accesses in caches and the main memory

The analogy fails with respect to the facts that:

- at the start of a working day the tools placed into the truck and the toolbox are preselected based on the experience gained on previous jobs, while in the case of a new program that is run on a computer there is no relevant data loaded into the caches before execution begins

- most of the tools in the toolbox and the truck are useful in successive jobs, while the data left in a cache by one program are not useful for the subsequent programs
  - tools displaced by the need to use other tools are never thrown away, while data in the cache blocks are simply overwritten if the blocks are not flagged as dirty
- 5.21. Each 32-bit number comprises 4 bytes. Hence, each page holds 1024 numbers. There is space for 256 pages in the 1M-byte portion of the main memory that is allocated for storing data during the computation.
- (a) Each column is one page; there will be 1024 page faults.
- (b) Processing of entire columns, one at a time, would be very inefficient and slow. However, if only one quarter of each column (for all columns) is processed before the next quarter is brought in from the disk, then each element of the array must be loaded into the memory twice. In this case, the number of page faults would be 2048.
- (c) Assuming that the computation time needed to normalize the numbers is negligible compared to the time needed to bring a page from the disk:
- Total time for (a) is  $1024 \times 40 \text{ ms} = 41 \text{ s}$
- Total time for (b) is  $2048 \times 40 \text{ ms} = 82 \text{ s}$
- 5.22. The operating system may increase the main memory pages allocated to a program that has a large number of page faults, using space previously allocated to a program with a few page faults.
- 5.23. Continuing the execution of an instruction interrupted by a page fault requires saving the entire state of the processor, which includes saving all registers that may have been affected by the instruction as well as the control information that indicates how far the execution has progressed. The alternative of re-executing the instruction from the beginning requires a capability to reverse any changes that may have been caused by the partial execution of the instruction.
- 5.24. The problem is that a page fault may occur during intermediate steps in the execution of a single instruction. The page containing the referenced location must be transferred from the disk into the main memory before execution can proceed. Since the time needed for the page transfer (a disk operation) is very long, as compared to instruction execution time, a context-switch will usually be made. (A context-switch consists of preserving the state of the currently executing program, and "switching" the processor to the execution of another program that is resident in the main memory.) The page transfer, via DMA, takes place while this other program executes. When the page transfer is complete, the original program can be resumed.
- Therefore, one of two features are needed in a system where the execution of an individual instruction may be suspended by a page fault. The first possibility

is to save the state of instruction execution. This involves saving more information (temporary programmer-transparent registers, etc.) than needed when a program is interrupted between instructions. The second possibility is to "unwind" the effects of the portion of the instruction completed when the page fault occurred, and then execute the instruction from the beginning when the program is resumed.

- 5.25. (a) The maximum number of bytes that can be stored on this disk is  $24 \times 14000 \times 400 \times 512 = 68.8 \times 10^9$  bytes.
- (b) The data transfer rate is  $(400 \times 512 \times 7200)/60 = 24.58 \times 10^6$  bytes/s.
- (c) Need 9 bits to identify a sector, 14 bits for a track, and 5 bits for a surface. Thus, a possible scheme is to use address bits  $A_{8-0}$  for sector,  $A_{22-9}$  for track, and  $A_{27-23}$  for surface identification. Bits  $A_{31-28}$  are not used.
- 5.26. The average seek time and rotational delay are 6 and 3 ms, respectively. The average data transfer rate from a track to the data buffer in the disk controller is 34 Mbytes/s. Hence, it takes  $8K/34M = 0.23$  ms to transfer a block of data.
- (a) The total time needed to access each block is  $9 + 0.23 = 9.23$  ms. The portion of time occupied by seek and rotational delay is  $9/9.23 = 0.97 = 97\%$ .
- (b) Only rotational delays are involved in 90% of the cases. Therefore, the average time to access a block is  $0.9 \times 3 + 0.1 \times 9 + 0.23 = 3.89$  ms. The portion of time occupied by seek and rotational delay is  $3.6/3.89 = 0.92 = 92\%$ .
- 5.27. (a) The rate of transfer to or from any one disk is 30 megabytes per second. Maximum memory transfer rate is  $4/(10 \times 10^{-9}) = 400 \times 10^6$  bytes/s, which is 400 megabytes per second. Therefore, 13 disks can be simultaneously flowing data to/from the main memory.
- (b)  $8K/30M = 0.27$  ms is needed to transfer 8K bytes to/from the disk. Seek and rotational delays are 6 ms and 3 ms, respectively. Therefore,  $8K/4 = 2K$  words are transferred in 9.27 ms. But in 9.27 ms there are  $(9.27 \times 10^{-3})/(0.01 \times 10^{-6}) = 927 \times 10^3$  memory (word) cycles available. Therefore, over a long period of time, any one disk steals only  $(2/927) \times 100 = 0.2\%$  of available memory cycles.
- 5.28. The sector size should influence the choice of page size, because the sector is the smallest directly addressable block of data on the disk that is read or written as a unit. Therefore, pages should be some small integral number of sectors in size.
- 5.29. The next record,  $j$ , to be accessed after a forward read of record  $i$  has just been completed might be in the forward direction, with probability 0.5 (4 records distance to the beginning of  $j$ ), or might be in the backward direction with probability 0.5 (6 records distance to the beginning of  $j$  plus 2 direction reversals). Time to scan over one record and an interrecord gap is

$$\frac{1}{800} \frac{\text{s}}{\text{cm}} \times \frac{1}{2000} \frac{\text{cm}}{\text{bit}} \times 4000 \text{ bits} \times 1000 \text{ ms} + 3 = 2.5 + 3 = 5.5 \text{ ms}$$

Therefore, average access and read time is

$$0.5(4 \times 5.5) + 0.5(6 \times 5.5 + 2 \times 225) + 5.5 = 258 \text{ ms}$$

If records can be read while moving in both directions, average access and read time is

$$0.5(4 \times 5.5) + 0.5(5 \times 5.5 + 225) + 5.5 = 142.75 \text{ ms}$$

Therefore, the average percentage gain is  $(258 - 142.75)/258 \times 100 = 44.7\%$   
The major gain is because the records being read are relatively close together, and one less direction reversal is needed.

# Chapter 6 – Arithmetic

6.1. Overflow cases are specifically indicated. In all other cases, no overflow occurs.

|                                                                                       |                                                                 |                                                                                       |                                                                 |                                                                    |                                                               |
|---------------------------------------------------------------------------------------|-----------------------------------------------------------------|---------------------------------------------------------------------------------------|-----------------------------------------------------------------|--------------------------------------------------------------------|---------------------------------------------------------------|
| $\begin{array}{r} 010110 \\ + 001001 \\ \hline 011111 \end{array}$                    | $\begin{array}{r} (+22) \\ + (+9) \\ \hline (+31) \end{array}$  | $\begin{array}{r} 101011 \\ + 100101 \\ \hline 010000 \\ \text{overflow} \end{array}$ | $\begin{array}{r} (-21) \\ + (-27) \\ \hline (-48) \end{array}$ | $\begin{array}{r} 111111 \\ + 000111 \\ \hline 000110 \end{array}$ | $\begin{array}{r} (-1) \\ + (+7) \\ \hline (+6) \end{array}$  |
| $\begin{array}{r} 011001 \\ + 010000 \\ \hline 101001 \\ \text{overflow} \end{array}$ | $\begin{array}{r} (+25) \\ + (+16) \\ \hline (+41) \end{array}$ | $\begin{array}{r} 110111 \\ + 111001 \\ \hline 110000 \end{array}$                    | $\begin{array}{r} (-9) \\ + (-7) \\ \hline (-16) \end{array}$   | $\begin{array}{r} 010101 \\ + 101011 \\ \hline 000000 \end{array}$ | $\begin{array}{r} (+21) \\ + (-21) \\ \hline (0) \end{array}$ |
| $\begin{array}{r} 010110 \\ - 011111 \\ \hline \end{array}$                           | $\begin{array}{r} (+22) \\ - (+31) \\ \hline (-9) \end{array}$  | $\begin{array}{r} 010110 \\ + 100001 \\ \hline 110111 \end{array}$                    |                                                                 |                                                                    |                                                               |
| $\begin{array}{r} 111110 \\ - 100101 \\ \hline \end{array}$                           | $\begin{array}{r} (-2) \\ - (-27) \\ \hline (+25) \end{array}$  | $\begin{array}{r} 111110 \\ + 011011 \\ \hline 011001 \end{array}$                    |                                                                 |                                                                    |                                                               |
| $\begin{array}{r} 100001 \\ - 011101 \\ \hline \end{array}$                           | $\begin{array}{r} (-31) \\ - (+29) \\ \hline (-60) \end{array}$ | $\begin{array}{r} 100001 \\ + 100011 \\ \hline 000100 \\ \text{overflow} \end{array}$ |                                                                 |                                                                    |                                                               |
| $\begin{array}{r} 111111 \\ - 000111 \\ \hline \end{array}$                           | $\begin{array}{r} (-1) \\ - (+7) \\ \hline (-8) \end{array}$    | $\begin{array}{r} 111111 \\ + 111001 \\ \hline 111000 \end{array}$                    |                                                                 |                                                                    |                                                               |
| $\begin{array}{r} 000111 \\ - 111000 \\ \hline \end{array}$                           | $\begin{array}{r} (+7) \\ - (-8) \\ \hline (+15) \end{array}$   | $\begin{array}{r} 000111 \\ + 001000 \\ \hline 001111 \end{array}$                    |                                                                 |                                                                    |                                                               |
| $\begin{array}{r} 011010 \\ - 100010 \\ \hline \end{array}$                           | $\begin{array}{r} (+26) \\ - (-30) \\ \hline (+56) \end{array}$ | $\begin{array}{r} 011010 \\ + 011110 \\ \hline 111000 \\ \text{overflow} \end{array}$ |                                                                 |                                                                    |                                                               |

6.2. (a) In the following answers, rounding has been used as the truncation method (see Section 6.7.3) when the answer cannot be represented exactly in the signed 6-bit format.

|         |        |                    |
|---------|--------|--------------------|
| 0.5:    | 010000 | all cases          |
| −0.123: | 100100 | Sign-and-magnitude |
|         | 111011 | 1's-complement     |
|         | 111100 | 2's-complement     |
| −0.75:  | 111000 | Sign-and-magnitude |
|         | 100111 | 1's-complement     |
|         | 101000 | 2's-complement     |
| −0.1:   | 100011 | Sign-and-magnitude |
|         | 111100 | 1's-complement     |
|         | 111101 | 2's-complement     |

(b)

$e = 2^{-6}$  (assuming rounding, as in (a))

$e = 2^{-5}$  (assuming chopping or Von Neumann rounding)

(c) assuming rounding:

|     |    |
|-----|----|
| (a) | 3  |
| (b) | 6  |
| (c) | 9  |
| (d) | 19 |

6.3. The two ternary representations are given as follows:

| Sign-and-magnitude | 3's-complement |
|--------------------|----------------|
| +11011             | 011011         |
| −10222             | 212001         |
| +2120              | 002120         |
| −1212              | 221011         |
| +10                | 000010         |
| −201               | 222022         |



6.4. Ternary numbers with addition and subtraction operations:

| Decimal<br>Sign-and-magnitude | Ternary<br>Sign-and-magnitude | Ternary<br>3's-complement |
|-------------------------------|-------------------------------|---------------------------|
| 56                            | +2002                         | 002002                    |
| -37                           | -1101                         | 221122                    |
| 122                           | 11112                         | 011112                    |
| -123                          | -11120                        | 211110                    |

Addition operations:

|              |              |              |
|--------------|--------------|--------------|
| 002002       | 002002       | 002002       |
| + 221122     | + 011112     | + 211110     |
| <hr/> 000201 | <hr/> 020121 | <hr/> 220112 |
| 221122       | 221122       | 011112       |
| + 011112     | + 211110     | + 211110     |
| <hr/> 010011 | <hr/> 210002 | <hr/> 222222 |

Subtraction operations:

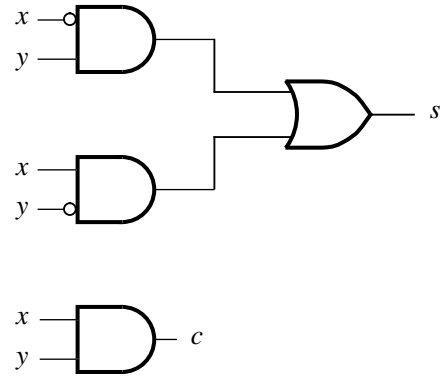
|              |          |
|--------------|----------|
| 002002       | 002002   |
| - 221122     | + 001101 |
| <hr/> 010110 |          |
| 002002       | 002002   |
| - 011112     | + 211111 |
| <hr/> 220120 |          |
| 002002       | 002002   |
| - 211110     | + 011120 |
| <hr/> 020122 |          |
| 221122       | 221122   |
| - 011112     | + 211111 |
| <hr/> 210010 |          |
| 221122       | 221122   |
| - 211110     | + 011120 |
| <hr/> 010012 |          |
| 011112       | 011112   |
| - 211110     | + 011120 |
| <hr/> 100002 |          |
|              | overflow |

6.5. (a)

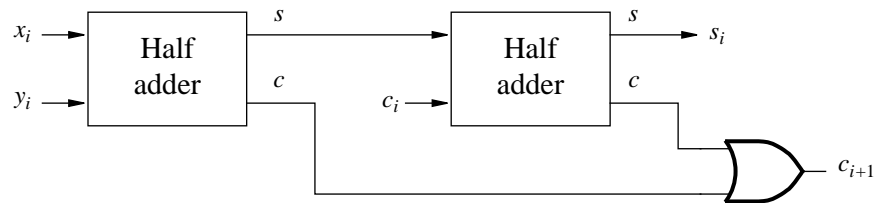
| $x$ | $y$ | $s$ | $c$ |
|-----|-----|-----|-----|
| 0   | 0   | 0   | 0   |
| 0   | 1   | 1   | 0   |
| 1   | 0   | 1   | 0   |
| 1   | 1   | 0   | 1   |

$$s = x \oplus y$$

$$c = xy$$



(b)



(c) The longest path through the circuit in Part (b) is 6 gate delays (including input inversions) in producing  $s_i$ ; and the longest path through the circuit in Figure 6.2a is 3 gate delays in producing  $s_i$ , assuming that  $s_i$  is implemented as a two-level AND-OR circuit, and including input inversions.

- 6.6. Assume that the binary integer is in memory location `BINARY`, and the string of bytes representing the answer starts at memory location `DECIMAL`, high-order digits first.

**68000 Program:**

|                   |                     |                             |                                                                              |
|-------------------|---------------------|-----------------------------|------------------------------------------------------------------------------|
|                   | <code>MOVE</code>   | <code>#10,D2</code>         |                                                                              |
|                   | <code>CLR.L</code>  | <code>D1</code>             |                                                                              |
|                   | <code>MOVE</code>   | <code>BINARY,D1</code>      | Get binary number;<br>note that high-order<br>word in D1 is still zero.      |
|                   | <code>MOVE.B</code> | <code>#4,D3</code>          | Use D3 as counter.                                                           |
| <code>LOOP</code> | <code>DIVU</code>   | <code>D2,D1</code>          | Leaves quotient in<br>low half of D1 and<br>remainder in high half<br>of D1. |
|                   | <code>SWAP</code>   | <code>D1</code>             |                                                                              |
|                   | <code>MOVE.B</code> | <code>D1,DECIMAL(D3)</code> |                                                                              |
|                   | <code>CLR</code>    | <code>D1</code>             | Clears low half of D1.                                                       |
|                   | <code>SWAP</code>   | <code>D1</code>             |                                                                              |
|                   | <code>DBRA</code>   | <code>D3,LOOP</code>        |                                                                              |

**IA-32 Program:**

|                         |                   |                             |                                                                        |
|-------------------------|-------------------|-----------------------------|------------------------------------------------------------------------|
|                         | <code>MOV</code>  | <code>EBX,10</code>         |                                                                        |
|                         | <code>MOV</code>  | <code>EAX,BINARY</code>     | Get binary number.                                                     |
|                         | <code>LEA</code>  | <code>EDI,DECIMAL</code>    |                                                                        |
|                         | <code>DEC</code>  | <code>EDI</code>            |                                                                        |
|                         | <code>MOV</code>  | <code>ECX,5</code>          | Load counter ECX.                                                      |
| <code>LOOPSTART:</code> | <code>DIV</code>  | <code>EBX</code>            | <code>[EAX]/[EBX]</code> ; quotient<br>in EAX and remainder<br>in EDX. |
|                         | <code>MOV</code>  | <code>[EDI + ECX],DL</code> |                                                                        |
|                         | <code>LOOP</code> | <code>LOOPSTART</code>      |                                                                        |

6.7. The ARM and IA-32 subroutines both use the following algorithm to convert the four-digit decimal integer  $D_3D_2D_1D_0$  (each  $D_i$  is in BCD code) into binary:

- Move  $D_0$  into register REG.
- Multiply  $D_1$  by 10.
- Add product into REG.
- Multiply  $D_2$  by 100.
- Add product into REG.
- Multiply  $D_3$  by 1000.
- Add product into REG.

(i) The ARM subroutine assumes that the addresses DECIMAL and BINARY are passed to it on the processor stack in positions param1 and param2 as shown in Figure 3.13. The subroutine first saves registers and sets up the frame pointer FP (R12).

**ARM Subroutine:**

|         |       |                   |                         |
|---------|-------|-------------------|-------------------------|
| CONVERT | STMFD | SP!,{R0–R6,FP,LR} | Save registers.         |
|         | ADD   | FP,SP,#28         | Load frame pointer.     |
|         | LDR   | R0,[FP,#8]        | Load R0 and R1          |
|         | LDR   | R0,[R0]           | with decimal digits.    |
|         | MOV   | R1,R0             |                         |
|         | AND   | R0,R0,#&F         | [R0] = $D_0$ .          |
|         | MOV   | R2,#&F            | Load mask bits into R2. |
|         | MOV   | R4,#10            | Load multipliers        |
|         | MOV   | R5,#100           | into R4, R5, and R6.    |
|         | MOV   | R6,#1000          |                         |
|         | AND   | R3,R2,R1,LSR #4   | Get $D_1$ into R3.      |
|         | MLA   | R0,R3,R4,R0       | Add $10D_1$ into R0.    |
|         | AND   | R3,R2,R1,LSR #8   | Get $D_2$ into R3.      |
|         | MLA   | R0,R3,R5,R0       | Add $100D_2$ into R0.   |
|         | AND   | R3,R2,R1,LSR #12  | Get $D_3$ into R3.      |
|         | MLA   | R0,R3,R6,R0       | Add $1000D_3$ into R0.  |
|         | LDR   | R1,[FP,#12]       | Store converted value   |
|         | STR   | R0,[R1]           | into BINARY.            |
|         | LDMFD | SP!,{R0–R6,FP,PC} | Restore registers       |
|         |       |                   | and return.             |

(ii) The IA-32 subroutine assumes that the addresses DECIMAL and BINARY are passed to it on the processor stack in positions param1 and param2 as shown in Figure 3.48. The subroutine first sets up the frame pointer EBP, and then allocates and initializes the local variables 10, 100, and 1000, on the stack.

#### IA-32 Subroutine:

|          |      |                |                                    |
|----------|------|----------------|------------------------------------|
| CONVERT: | PUSH | EBP            | Set up frame                       |
|          | MOV  | EBP,ESP        | pointer.                           |
|          | PUSH | 10             | Allocate and initialize            |
|          | PUSH | 100            | local variables.                   |
|          | PUSH | 1000           |                                    |
|          | PUSH | EDX            | Save registers.                    |
|          | PUSH | ESI            |                                    |
|          | PUSH | EAX            |                                    |
|          | MOV  | EDX,[EBP + 8]  | Load four decimal                  |
|          | MOV  | EDX,[EDX]      | digits into                        |
|          | MOV  | ESI,EDX        | EDX and ESI.                       |
|          | AND  | EDX,FH         | [EDX] = $D_0$ .                    |
|          | SHR  | ESI,4          |                                    |
|          | MOV  | EAX,ESI        |                                    |
|          | AND  | EAX,FH         |                                    |
|          | MUL  | [EBP - 4]      |                                    |
|          | ADD  | EDX,EAX        | [EDX] = binary of $D_1D_0$ .       |
|          | SHR  | ESI,4          |                                    |
|          | MOV  | EAX,ESI        |                                    |
|          | AND  | EAX,FH         |                                    |
|          | MUL  | [EBP - 8]      |                                    |
|          | ADD  | EDX,EAX        | [EDX] = binary of $D_2D_1D_0$ .    |
|          | SHR  | ESI,4          |                                    |
|          | MOV  | EAX,ESI        |                                    |
|          | AND  | EAX,FH         |                                    |
|          | MUL  | [EBP - 12]     |                                    |
|          | ADD  | EDX,EAX        | [EDX] = binary of $D_3D_2D_1D_0$ . |
|          | MOV  | EAX,[EBP + 12] | Store converted                    |
|          | MOV  | [EAX],EDX      | value into BINARY.                 |
|          | POP  | EAX            | Restore registers.                 |
|          | POP  | ESI            |                                    |
|          | POP  | EDX            |                                    |
|          | ADD  | ESP,12         | Remove local parameters.           |
|          | POP  | EBP            | Restore EBP.                       |
|          | RET  |                | Return.                            |

(iii) The 68000 subroutine uses a loop structure to convert the four-digit decimal integer  $D_3D_2D_1D_0$  (each  $D_i$  is in BCD code) into binary. At the end of successive passes through the loop, register D0 contains the accumulating values  $D_3$ ,  $10D_3 + D_2$ ,  $100D_3 + 10D_2 + D_1$ , and  $\text{binary} = 1000D_3 + 100D_2 + 10D_1 + D_0$ . Assume that DECIMAL is the address of a 16-bit word containing the four BCD digits, and that BINARY is the address of a 16-bit word that is to contain the converted binary value.

The addresses DECIMAL and BINARY are passed to the subroutine in registers A0 and A1.

**68000 Subroutine:**

|         |         |             |                                                                         |
|---------|---------|-------------|-------------------------------------------------------------------------|
| CONVERT | MOVEM.L | D0–D2,–(A7) | Save registers.                                                         |
|         | CLR.L   | D0          |                                                                         |
|         | CLR.L   | D1          |                                                                         |
|         | MOVE.W  | (A0),D1     | Load four decimal digits into D1.                                       |
| LOOP    | MOVE.B  | #3,D2       | Load counter D3.                                                        |
|         | MULU.W  | #10,D0      | Multiply accumulated value in D0 by 10.                                 |
|         | ASL.L   | #4,D1       | Bring next $D_i$ digit into low half of D1.                             |
|         | SWAP.W  | D1          | into low half of D1.                                                    |
|         | ADD.W   | D1,D0       | Add into accumulated value in D0.                                       |
|         | CLR.W   | D1          | Clear out current digit and bring remaining digits into low half of D1. |
|         | SWAP.W  | D1          |                                                                         |
|         | DBRA    | D2,LOOP     | Check if done.                                                          |
|         | MOVE.W  | D0,(A1)     | Store binary result in BINARY.                                          |
|         | MOVEM.L | (A7)+,D0–D2 | Restore registers.                                                      |
|         | RTS     |             | Return.                                                                 |

6.8. (a) The output carry is 1 when  $A + B \geq 10$ . This is the condition that requires the further addition of  $6_{10}$ .

(b)

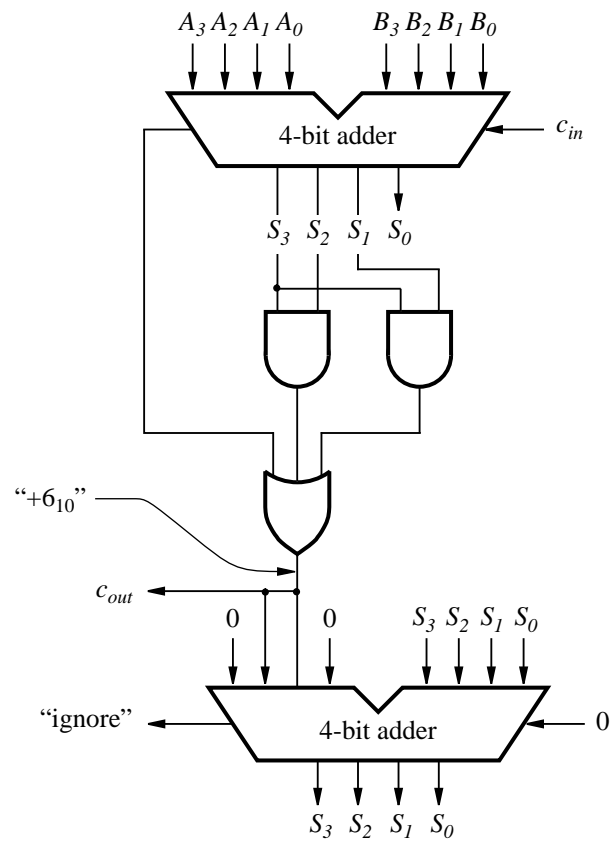
$$(1) \quad \begin{array}{r} 0101 \\ + 0110 \\ \hline 1011 \end{array} > 10_{10} \quad \begin{array}{r} 5 \\ + 6 \\ \hline 11 \end{array}$$

$$\begin{array}{r} + 0110 \\ \hline 0001 \end{array}$$

output carry = 1

$$(2) \quad \begin{array}{r} 0011 \\ + 0100 \\ \hline 0111 \end{array} < 10_{10} \quad \begin{array}{r} 3 \\ + 4 \\ \hline 7 \end{array}$$

(c)



6.9. Consider the truth table in Figure 6.1 for the case  $i = n - 1$ , that is, for the sign bit position. Overflow occurs only when  $x_{n-1}$  and  $y_{n-1}$  are the same and  $s_{n-1}$  is different. This occurs in the second and seventh rows of the table; and  $c_n$  and  $c_{n-1}$  are different only in those rows. Therefore,  $c_n \oplus c_{n-1}$  is a correct indicator of overflow.

6.10. (a) The additional logic is defined by the logic expressions:

$$\begin{aligned}
 c_{16} &= G_0^{II} + P_0^{II} c_0 \\
 c_{32} &= G_1^{II} + P_1^{II} G_0^{II} + P_1^{II} P_0^{II} c_0 \\
 c_{48} &= G_2^{II} + P_2^{II} G_1^{II} + P_2^{II} P_1^{II} G_0^{II} + P_2^{II} P_1^{II} P_0^{II} c_0 \\
 c_{64} &= G_3^{II} + P_3^{II} G_2^{II} + P_3^{II} P_2^{II} G_1^{II} + P_3^{II} P_2^{II} P_1^{II} G_0^{II} + P_3^{II} P_2^{II} P_1^{II} P_0^{II} c_0
 \end{aligned}$$

This additional logic is identical in form to the logic inside the lookahead circuit in Figure 6.5. (Note that the outputs  $c_{16}$ ,  $c_{32}$ ,  $c_{48}$ , and  $c_{64}$ , produced by the 16-bit adders are not needed because those outputs are produced by the additional logic.)

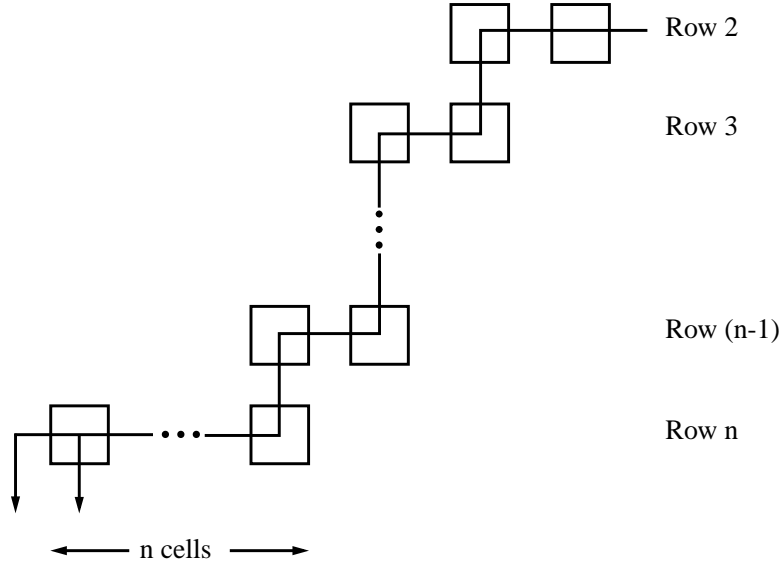
(b) The inputs  $G_i^{II}$  and  $P_i^{II}$  to the additional logic are produced after 5 gate delays, the same as the delay for  $c_{16}$  in Figure 6.5. Then all outputs from the additional logic, including  $c_{64}$ , are produced 2 gate delays later, for a total of 7 gate delays. The carry input  $c_{48}$  to the last 16-bit adder is produced after 7 gate delays. Then  $c_{60}$  into the last 4-bit adder is produced after 2 more gate delays, and  $c_{63}$  is produced after another 2 gate delays inside that 4-bit adder. Finally, after one more gate delay (an XOR gate),  $s_{63}$  is produced with a total of  $7 + 2 + 2 + 1 = 12$  gate delays.

(c) The variables  $s_{31}$  and  $c_{32}$  are produced after 12 and 7 gate delays, respectively, in the 64-bit adder. These two variables are produced after 10 and 7 gate delays in the 32-bit adder, as shown in Section 6.2.1.



- 6.11. (a) Each B cell requires 3 gates as shown in Figure 6.4a. The carries  $c_1$ ,  $c_2$ ,  $c_3$ , and  $c_4$ , require 2, 3, 4, and 5, gates, respectively; and the outputs  $G_0^I$  and  $P_0^I$  require 4 and 1 gates, as seen from the logic expressions in Section 6.2.1. Therefore, a total of  $12 + 19 = 31$  gates are required for the 4-bit adder.
- (b) Four 4-bit adders require  $4 \times 31 = 124$  gates, and the carry-lookahead logic block requires 19 gates because it has the same structure as the lookahead block in Figure 6.4. Total gate count is thus 143. However, we should subtract  $4 \times 5 = 20$  gates from this total corresponding to the logic for  $c_4$ ,  $c_8$ ,  $c_{12}$ , and  $c_{16}$ , that is in the 4-bit adders but which is replaced by the lookahead logic in Figure 6.5. Therefore, total gate count for the 16-bit adder is  $143 - 20 = 123$  gates.

6.12. The worst case delay path is shown in the following figure:



Each of the two FA blocks in rows 2 through  $n - 1$  introduces 2 gate delays, for a total of  $4(n - 2)$  gate delays. Row  $n$  introduces  $2n$  gate delays. Adding in the initial AND gate delay for row 1 and all other cells, total delay is:

$$4(n - 2) + 2n + 1 = 6n - 8 + 1 = 6(n - 1) - 1$$

6.13. The solutions, including decimal equivalent checks, are:

$$\begin{array}{rcl}
 \text{B} & = & 00101 \quad (5) \\
 \times \text{A} & = & \begin{array}{r} 10101 \\ 00101 \\ 0 \\ 00101 \\ 001010 \\ \hline 001101001 \end{array} \quad \begin{array}{r} \times (21) \\ (105) \\ \\ \\ (105) \end{array}
 \end{array}$$

$$\begin{array}{rcl}
 & & 100 \\
 00101 & \sqrt{10101} & \\
 & \underline{101} & \\
 & 00001 &
 \end{array}
 \quad
 \begin{array}{rcl}
 & & 4 \\
 5 & \sqrt{21} & \\
 & \underline{20} & \\
 & 1 &
 \end{array}$$

6.14. The multiplication and division charts are:

$A \times B :$

|   |         |       |                       |
|---|---------|-------|-----------------------|
|   | M       |       |                       |
|   | 00101   |       |                       |
| 0 | 00000   | 10101 | Initial configuration |
| C | A       | Q     |                       |
| 0 | 00101   | 10101 | 1st cycle             |
| 0 | 00010   | 11010 |                       |
| 0 | 00010   | 11010 | 2nd cycle             |
| 0 | 00001   | 01101 |                       |
| 0 | 00110   | 01101 | 3rd cycle             |
| 0 | 00011   | 00110 |                       |
| 0 | 00011   | 00110 | 4th cycle             |
| 0 | 00001   | 10011 |                       |
| 0 | 00110   | 10011 | 5th cycle             |
| 0 | 00011   | 01001 |                       |
|   | product |       |                       |

$A / B :$

|          |           |          |                       |
|----------|-----------|----------|-----------------------|
|          | 000000    | 10101    |                       |
|          | A         | Q        |                       |
|          | 000101    |          | Initial configuration |
|          | M         |          |                       |
| shift    | 000001    | 0 1 0 1  |                       |
| subtract | 111011    |          | 1st cycle             |
|          | 111100    | 0 1 0 1  |                       |
| shift    | 111000    | 1 0 1    |                       |
| add      | 000101    |          | 2nd cycle             |
|          | 111101    | 1 0 1    |                       |
| shift    | 111011    | 0 1      |                       |
| add      | 000101    |          | 3rd cycle             |
|          | 000000    | 0 1      |                       |
| shift    | 000000    | 1        |                       |
| subtract | 111011    |          | 4th cycle             |
|          | 111011    | 1        |                       |
| shift    | 110111    | 0 0 1    |                       |
| add      | 000101    |          | 5th cycle             |
|          | 111100    | 0 0 1    |                       |
| add      | 000101    |          |                       |
|          | 000001    | quotient |                       |
|          | remainder |          |                       |

### 6.15. ARM Program:

Use R0 as the loop counter.

|      |       |           |                              |
|------|-------|-----------|------------------------------|
|      | MOV   | R1,#0     |                              |
|      | MOV   | R0,#32    |                              |
| LOOP | TST   | R2,#1     | Test LSB of multiplier.      |
|      | ADDNE | R1,R3,R1  | Add multiplicand if LSB = 1. |
|      | MOV   | R1,R1,RRX | Shift [R1] and [R2] right    |
|      | MOV   | R2,R2,RRX | one bit position, with [C].  |
|      | SUBS  | R0,R0,#1  | Check if done.               |
|      | BGT   | LOOP      |                              |

### 68000 program:

Assume that D2 and D3 contain the multiplier and the multiplicand, respectively. The high- and low-order halves of the product will be stored in D1 and D2. Use D0 as the loop counter.

|       |        |         |                              |
|-------|--------|---------|------------------------------|
|       | CLR.L  | D1      |                              |
|       | MOVE.B | #31,D0  |                              |
| LOOP  | ANDI.W | #1,D2   | Test LSB of multiplier.      |
|       | BEQ    | NOADD   |                              |
|       | ADD.L  | D3,D1   | Add multiplicand if LSB = 1. |
| NOADD | ROXR.L | #1,D1   | Shift [D1] and [D2] right    |
|       | ROXR.L | #1,D2   | one bit position, with [C].  |
|       | DBRA   | D0,LOOP | Check if done.               |

### IA-32 Program:

Use registers EAX, EDX, and EDI, as  $R_1$ ,  $R_2$ , and  $R_3$ , respectively, and use ECX as the loop counter.

|            |      |           |                               |
|------------|------|-----------|-------------------------------|
|            | MOV  | EAX,0     |                               |
|            | MOV  | ECX,32    |                               |
|            | SHR  | EDX,1     | Set [CF] = LSB of multiplier. |
| LOOPSTART: | JNC  | NOADD     |                               |
|            | ADD  | EAX,EDI   | Add multiplicand if LSB = 1.  |
| NOADD:     | RCR  | EAX,1     | Shift [EAX] and [EDX] right   |
|            | RCR  | EDX,1     | one bit position, with [CF].  |
|            | LOOP | LOOPSTART | Check if done.                |

#### 6.16. ARM Program:

Use the register assignment R1, R2, and R0, for the dividend, divisor, and remainder, respectively. As computation proceeds, the quotient will be shifted into R1.

|      |        |              |                        |
|------|--------|--------------|------------------------|
|      | MOV    | R0,#0        | Clear R0.              |
|      | MOV    | R3,#32       | Initialize counter R3. |
| LOOP | MOVS   | R1,R1,LSL #1 | Two-register left      |
|      | ADCS   | R0,R0,R0     | shift of R0 and R1     |
|      |        |              | by one position.       |
|      | SUBCCS | R0,R0,R2     | Implement step 1       |
|      | ADDCSS | R0,R0,R2     | of the algorithm.      |
|      | ORRPL  | R1,R1,#1     |                        |
|      | SUBS   | R3,R3,#1     | Check if done.         |
|      | BGT    | LOOP         |                        |
|      | TST    | R0,R0        | Implement step 2       |
|      | ADDMI  | R0,R2,R0     | of the algorithm.      |

#### 68000 Program:

Assume that D1 and D2 contain the dividend and the divisor, respectively. We will use D0 to store the remainder. As computation proceeds, the quotient will be shifted into D1.

|       |        |         |                            |
|-------|--------|---------|----------------------------|
|       | CLR    | D0      | Clear D0.                  |
|       | MOVE.B | #15,D3  | Initialize counter D3.     |
| LOOP  | ASL    | #1,D1   | Two-register left shift of |
|       | ROXL   | #1,D0   | D0 and D1 by one position. |
|       | BCS    | NEGRM   | Implement step 1           |
|       | SUB    | D2,D0   | of the algorithm.          |
|       | BRA    | SETQ    |                            |
| NEGRM | ADD    | D2,D0   |                            |
| SETQ  | BMI    | COUNT   |                            |
|       | ORI    | #1,D1   |                            |
| COUNT | DBRA   | D3,LOOP | Check if done.             |
|       | TST    | D0      | Implement step 2           |
|       | BPL    | DONE    | of the algorithm.          |
|       | ADD    | D2,D0   |                            |
| DONE  | ...    |         |                            |

**IA-32 Program:**

Use the register assignment EAX, EBX, and EDX, for the dividend, divisor, and remainder, respectively. As computation proceeds, the quotient is shifted into EAX.

|            |      |           |                         |
|------------|------|-----------|-------------------------|
|            | MOV  | EDX,0     | Clear EDX.              |
|            | MOV  | ECX,32    | Initialize counter ECX. |
| LOOPSTART: | SHL  | EAX,1     | Two-register left       |
|            | RCL  | EDX,1     | shift of EDX and EAX    |
|            |      |           | by one position.        |
|            | JC   | NEGRM     | Implement step 1        |
|            | SUB  | EDX,EBX   | of the algorithm.       |
|            | JMP  | SETQ      |                         |
| NEGRM:     | ADD  | EDX,EBX   |                         |
| SETQ:      | JS   | COUNT     |                         |
|            | OR   | EAX,1     |                         |
| COUNT:     | LOOP | LOOPSTART | Check if done.          |
|            | TEST | EDX,EDX   | Implement step 2        |
|            | JNS  | DONE      | of the algorithm.       |
|            | ADD  | EDX,EBX   |                         |
| DONE:      | ...  |           |                         |

sign  
extension

sign  
extension

sign  
extension

6.18. The multiplication answers are:

(a) 
$$\begin{array}{r} 010111 \\ \times 110110 \\ \hline \end{array}$$

$$\begin{array}{r} 010111 \\ -1 \quad +2 \quad -2 \\ \hline 111111 \quad 10100010 \\ 0000 \quad 0101110 \\ \hline 111110210101 \\ \hline 1111100011010 \end{array}$$

(b) 
$$\begin{array}{r} 110011 \\ \times 101100 \\ \hline \end{array}$$

$$\begin{array}{r} 110011 \\ -1 \quad -1 \quad 0 \\ \hline 0000 \quad 0001101 \\ 0000 \quad 011101 \\ \hline 00010000100 \end{array}$$

(c) 
$$\begin{array}{r} 110101 \\ \times 011011 \\ \hline \end{array}$$

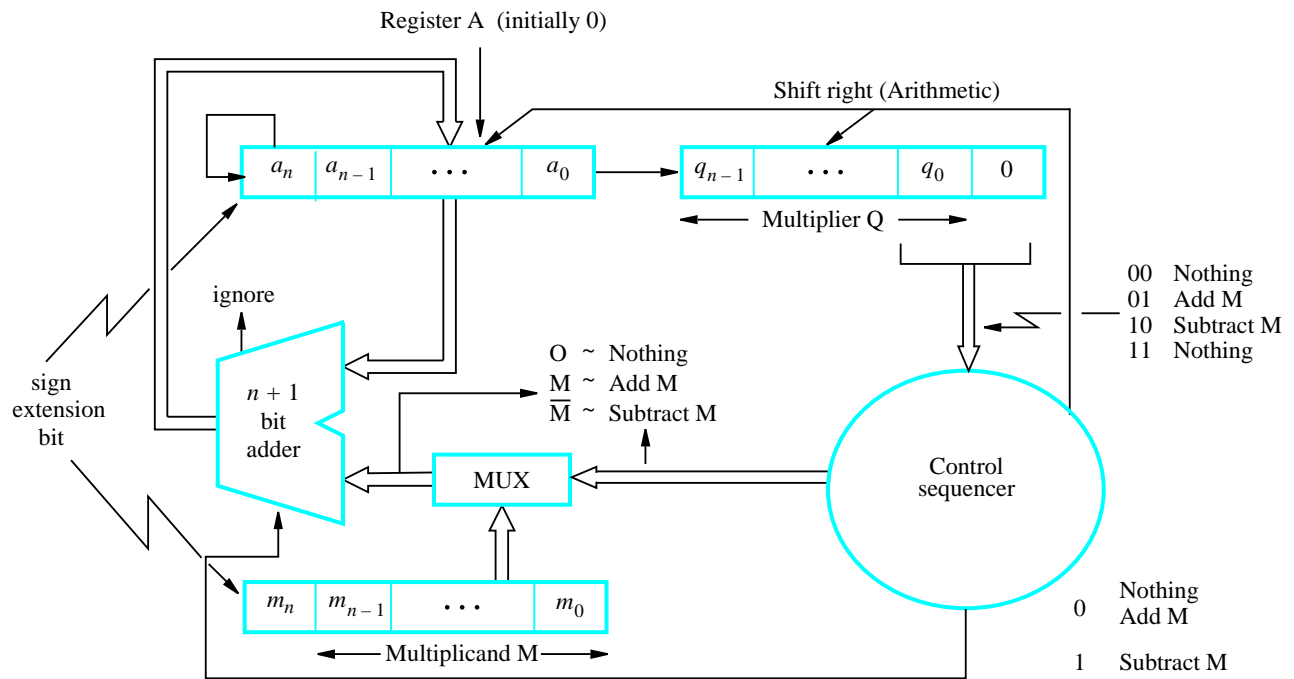
$$\begin{array}{r} 110101 \\ +2 \quad -1 \quad -1 \\ \hline 000000 \quad 0001011 \\ 0000 \quad 0001011 \\ \hline 1110101011 \\ \hline 111011010111 \end{array}$$

(d) 
$$\begin{array}{r} 001111 \\ \times 001111 \\ \hline \end{array}$$

$$\begin{array}{r} 001111 \\ +1 \quad -1 \\ \hline 111111 \quad 110001 \\ 00001111 \\ \hline 000011100001 \end{array}$$



- 6.19. Both the A and M registers are augmented by one bit to the left to hold a sign extension bit. The adder is changed to an  $n + 1$ -bit adder. A bit is added to the right end of the Q register to implement the Booth multiplier recoding operation. It is initially set to zero. The control logic decodes the two bits at the right end of the Q register according to the Booth algorithm, as shown in the following logic circuit. The right shift is an arithmetic right shift as indicated by the repetition of the extended sign bit at the left end of the A register. (The only case that actually requires the sign extension bit is when the  $n$ -bit multiplicand is the value  $-2^{(n-1)}$ ; for all other operands, the A and M registers could have been  $n$ -bit registers and the adder could have been an  $n$ -bit adder.)



6.20 (a)

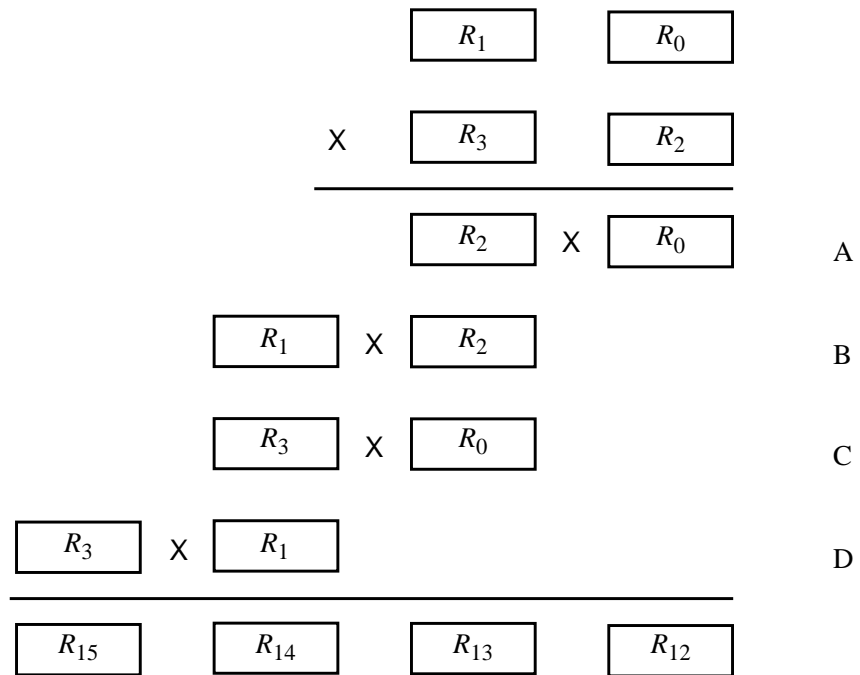
$$\begin{array}{r}
 1110 \\
 \times 1101 \\
 \hline
 1110 \\
 0000 \\
 1000 \\
 0000 \\
 \hline
 0110
 \end{array}
 \qquad
 \begin{array}{r}
 -2 \\
 \times -3 \\
 \hline
 6
 \end{array}$$

(b)

$$\begin{array}{r}
 0010 \\
 \times 1110 \\
 \hline
 0000 \\
 0100 \\
 1000 \\
 0000 \\
 \hline
 1100
 \end{array}
 \qquad
 \begin{array}{r}
 2 \\
 \times -2 \\
 \hline
 -4
 \end{array}$$

This technique works correctly for the same reason that modular addition can be used to implement signed-number addition in the 2's-complement representation, because multiplication can be interpreted as a sequence of additions of the multiplicand to shifted versions of itself.

- 6.21. The four 32-bit subproducts needed to generate the 64-bit product are labeled A, B, C, and D, and shown in their proper shifted positions in the following figure:



The 64-bit product is the sum of A, B, C, and D. Using register transfers and multiplication and addition operations executed by the arithmetic unit described, the 64-bit product is generated without using any extra registers by the following steps:

$$\begin{aligned}
R_{12} &\leftarrow [R_0] \\
R_{13} &\leftarrow [R_2] \\
R_{14} &\leftarrow [R_1] \\
R_{15} &\leftarrow [R_3] \\
R_3 &\leftarrow [R_{14}] \\
R_1 &\leftarrow [R_{15}] \\
R_{13}, R_{12} &\leftarrow [R_{13}] \times [R_{12}] \\
R_{15}, R_{14} &\leftarrow [R_{15}] \times [R_{14}] \\
R_3, R_2 &\leftarrow [R_3] \times [R_2] \\
R_1, R_0 &\leftarrow [R_1] \times [R_0] \\
R_{13} &\leftarrow [R_2] \text{ Add } [R_{13}] \\
R_{14} &\leftarrow [R_3] \text{ Add with carry } [R_{14}] \\
R_{15} &\leftarrow 0 \text{ Add with carry } [R_{15}] \\
R_{13} &\leftarrow [R_0] \text{ Add } [R_{13}] \\
R_{14} &\leftarrow [R_1] \text{ Add with carry } [R_{14}] \\
R_{15} &\leftarrow 0 \text{ Add with carry } [R_{15}]
\end{aligned}$$

This procedure destroys the original contents of the operand registers. Steps 5 and 6 result in swapping the contents of  $R_1$  and  $R_3$  so that subproducts B and C can be computed in adjacent register pairs. Steps 11, 12, and 13, add the subproduct B into the 64-bit product registers; and steps 14, 15, and 16, add the subproduct C into these registers.

- 6.22. (a) The worst case delay path in Figure 6.16a is along the staircase pattern that includes the two FA blocks at the right end of each of the first two rows (a total of four FA block delays), followed by the four FA blocks in the third row. Total delay is therefore 17 gate delays, including the initial AND gate delay to develop all bit products.

In Figure 6.16b, the worst case delay path is vertically through the first two rows (a total of two FA block delays), followed by the four FA blocks in the third row for a total of 13 gate delays, including the initial AND gate delay to develop all bit products.

(b) Both arrays are  $4 \times 4$  cases.

Note that 17 is the result of applying the expression  $6(n - 1) - 1$  with  $n = 4$  for the array in Figure 6.16a.

A similar expression for the Figure 6.16b array is developed as follows. The delay through  $(n - 2)$  carry-save rows of FA blocks is  $2(n - 2)$  gate delays, followed by  $2n$  gate delays along the  $n$  FA blocks of the last row, for a total of

$$2(n - 2) + 2n + 1 = 4(n - 1) + 1$$

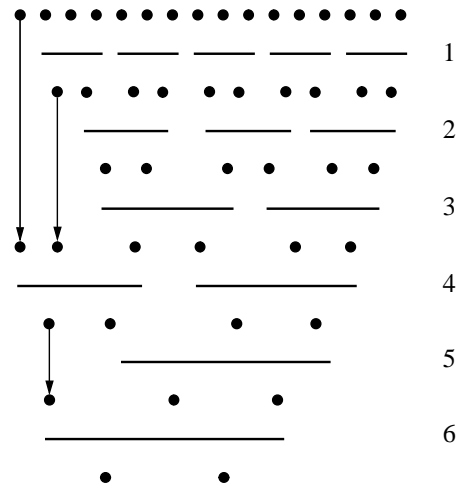
gate delays, including the initial AND gate delay to develop all bit products. The answer is thus 13, as computed directly in Part (a), for the  $4 \times 4$  case.

- 6.23. The number of reduction steps  $n$  to reduce  $k$  summands to 2 is given by  $k(2/3)^n = 2$ , because each step reduces 3 summands to 2. Then we have:

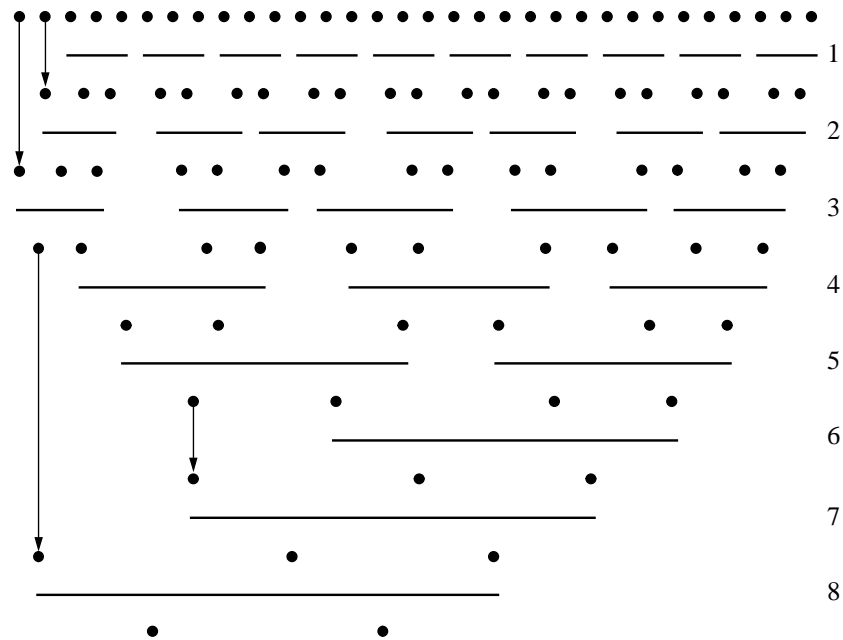
$$\begin{aligned} \log_2 k + n(\log_2 2 - \log_2 3) &= \log_2 2 \\ \log_2 k &= 1 + n(\log_2 3 - \log_2 2) \\ &= 1 + n(1.59 - 1) \\ n &= \frac{(\log_2 k) - 1}{0.59} \\ &= 1.7 \log_2 k - 1.7 \end{aligned}$$

This answer is only an approximation because the number of summands is not a multiple of 3 in each reduction step.

6.24. (a) Six CSA levels are needed:



(b) Eight CSA levels are needed:



(c) The approximation gives 5.1 and 6.8 CSA levels, compared to 6 and 8 from Parts (a) and (b).

6.25. (a)

|        |   |       |        |
|--------|---|-------|--------|
| +1.7   | 0 | 01111 | 101101 |
| -0.012 | 1 | 01000 | 100010 |
| +19    | 0 | 10011 | 001100 |
| 1/8    | 0 | 01100 | 000000 |

“Rounding” has been used as the truncation method in these answers.

(b) Other than exact 0 and  $\pm\infty$ , the smallest numbers are  $\pm 1.000000 \times 2^{-14}$  and the largest numbers are  $\pm 1.111111 \times 2^{15}$ .

(c) Assuming sign-and-magnitude format, the smallest and largest integers (other than 0) are  $\pm 1$  and  $\pm(2^{11} - 1)$ ; and the smallest and largest fractions (other than 0) are  $\pm 2^{-11}$  and approximately  $\pm 1$ .

(d)

$$\begin{aligned} A + B &= 0\ 10001\ 000000 \\ A - B &= 0\ 10001\ 110110 \\ A \times B &= 1\ 10010\ 001011 \\ A/B &= 1\ 10000\ 011011 \end{aligned}$$

“Rounding” has been used as the truncation method in these answers.

6.26. (a) Shift the mantissa of  $B$  right two positions, and tentatively set the exponent of the sum to 100001. Add mantissas:

$$\begin{array}{r} (A) \quad 1.1111111000 \\ (B) \quad 0.01001010101 \\ \hline 10.01001001101 \end{array}$$

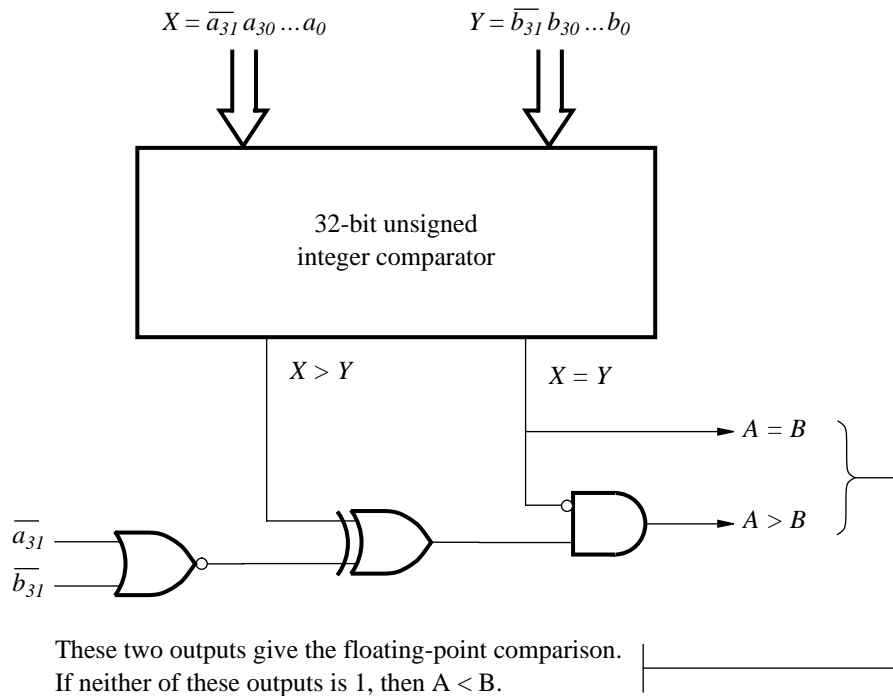
Shift right one position to put in normalized form: 1.001001001101 and increase exponent of sum to 100010. Truncate the mantissa to the right of the binary point to 9 bits by rounding to obtain 001001010. The answer is 0 100010 001001010.

(b)

$$\begin{aligned} \text{Largest} &\approx 2 \times 2^{31} \\ \text{Smallest} &\approx 1 \times 2^{-30} \end{aligned}$$

This assumes that the two end values, 63 and 0 in the excess-31 exponent, are used to represent infinity and exact 0, respectively.

- 6.27. Let  $A$  and  $B$  be two floating-point numbers. First, assume that  $S_A = S_B = 0$ . If  $E'_A > E'_B$ , considered as unsigned 8-bit numbers, then  $A > B$ . If  $E'_A = E'_B$ , then  $A > B$  if  $M_A > M_B$ . This means that  $A > B$  if the 31 bits after the sign in the representation for  $A$  is greater than the 31 bits representing  $B$ , when both are considered as integers. In the logic circuit shown below, all possibilities for the sum bit are also taken into account. In the circuit, let  $A = a_{31}a_{30} \dots a_0$  and  $B = b_{31}b_{30} \dots b_0$  be the two floating-point numbers to be compared.



- 6.28. Convert the given decimal mantissa into a binary floating-point number by using the integer facilities in the computer to implement the conversion algorithms in Appendix E. This will yield a floating-point number  $f_i$ . Then, using the computer's floating-point facilities, compute  $f_i \times t_i$ , as required.

- 6.29.  $(0.1)^{10} \Rightarrow (0.00011001100\dots)$

The signed, 8-bit approximations to this decimal number are:

|                       |                              |
|-----------------------|------------------------------|
| Chopping:             | $(0.1)_{10} = (0.0001100)_2$ |
| Von Neumann Rounding: | $(0.1)_{10} = (0.0001101)_2$ |
| Rounding:             | $(0.1)_{10} = (0.0001101)_2$ |



- 6.30. Consider  $A - B$ , where  $A$  and  $B$  are 6-bit (normalized) mantissas of floating-point numbers. Because of differences in exponents,  $B$  must be shifted 6 positions before subtraction.

$$\begin{aligned} A &= 0.100000 \\ B &= 0.100001 \end{aligned}$$

After shifting, we have:

$$\begin{array}{rcl} A &= & 0.100000\ 000 \\ -B &= & \begin{array}{r} 0.000000\ 101 \\ \hline 0.011111\ 011 \end{array} \leftarrow \text{sticky bit} \\ \text{normalize} && 0.111110\ 110 \\ \text{round} && 0.111111 \leftarrow \text{correct answer (rounded)} \end{array}$$

With only 2 guard bits, we would have had:

$$\begin{array}{rcl} A &= & 0.100000\ 00 \\ -B &= & \begin{array}{r} 0.000000\ 11 \\ \hline 0.011111\ 01 \end{array} \\ \text{normalize} && 0.111110\ 10 \\ \text{round} && 0.111110 \end{array}$$

- 6.31. The binary versions of the decimal fractions  $-0.123$  and  $-0.1$  are not exact. Using 3 guard bits, with the last bit being the sticky bit, the fractions  $0.123$  and  $0.1$  are represented as:

$$\begin{aligned} 0.123 &= 0.00011\ 111 \\ 0.1 &= 0.00011\ 001 \end{aligned}$$

The three representations for both fractions using each of the three truncation methods are:

|            |                    | Chop    | Von Neumann | Round   |
|------------|--------------------|---------|-------------|---------|
| $-0.123$ : | Sign-and-magnitude | 1.00011 | 1.00011     | 1.00100 |
|            | 1's-complement     | 1.11100 | 1.11100     | 1.11011 |
|            | 2's-complement     | 1.11101 | 1.11101     | 1.11100 |
| $-0.1$ :   | Sign-and-magnitude | 1.00011 | 1.00011     | 1.00011 |
|            | 1's-complement     | 1.11100 | 1.11100     | 1.11100 |
|            | 2's-complement     | 1.11101 | 1.11101     | 1.11101 |

6.32. The relevant truth table and logic equations are:

| ADD(0) /<br>SUBTRACT(1)<br>( $AS$ )  | $S_A$ | $S_B$ | sign from<br>8-bit<br>subtractor<br>( $8_s$ ) | sign from<br>25-bit adder/<br>subtractor<br>( $25_s$ ) | ADD/<br>SUB | $S_R$ |
|--------------------------------------|-------|-------|-----------------------------------------------|--------------------------------------------------------|-------------|-------|
| 0                                    | 0     | 0     | 0                                             | 0                                                      | 0           | 0     |
|                                      |       |       | 1                                             | 1                                                      |             | d     |
| 0                                    | 0     | 1     | 0                                             | 0                                                      | 1           | 0     |
|                                      |       |       | 1                                             | 1                                                      |             | d     |
| 0                                    | 1     | 0     | 0                                             | 0                                                      | 1           | 1     |
|                                      |       |       | 1                                             | 0                                                      |             | d     |
| 0                                    | 1     | 1     | 0                                             | 0                                                      | 0           | 1     |
|                                      |       |       | 1                                             | 1                                                      |             | d     |
| 1                                    | 0     | 0     | 0                                             | 0                                                      | 1           | 0     |
|                                      |       |       | 1                                             | 0                                                      |             | d     |
| 1                                    | 0     | 1     | 0                                             | 0                                                      | 0           | 0     |
|                                      |       |       | 1                                             | 1                                                      |             | d     |
| 1                                    | 1     | 0     | 0                                             | 0                                                      | 0           | 1     |
|                                      |       |       | 1                                             | 0                                                      |             | d     |
| 1                                    | 1     | 1     | 0                                             | 0                                                      | 1           | 0     |
|                                      |       |       | 1                                             | 0                                                      |             | d     |
| these variables<br>determine ADD/SUB |       |       | 1                                             | 1                                                      |             | d     |

| $S_A S_B$                      | 00 | 01 | 11 | 10 |
|--------------------------------|----|----|----|----|
| ADD(0)/<br>SUBTRACT(1)<br>(AS) | 0  | 0  | 1  | 0  |
|                                | 1  | 1  | 0  | 1  |

$ADD/SUB = AS \oplus S_A \oplus S_B$

| $S_B \ 8_s$ | 00 | 01 | 11 | 10 |
|-------------|----|----|----|----|
| 00          | 0  | 1  | 1  | 0  |
| 01          | 0  | 0  | 1  | 1  |
| 11          | 1  | 1  | 0  | 0  |
| 10          | 0  | 1  | 1  | 0  |

$25_s = 0$

| $S_B \ 8_s$ | 00 | 01 | 11 | 10 |
|-------------|----|----|----|----|
| 00          | d  | 0  | d  | 1  |
| 01          | d  | d  | d  | d  |
| 11          | d  | d  | d  | d  |
| 10          | 1  | d  | 0  | d  |

$25_s = 1$

$$S_R = 25_s \bar{S}_A + \bar{25}_s S_A \bar{8}_s + AS \bar{S}_B 8_s + \bar{AS} S_B 8_s$$

6.33. The largest that  $n$  can be is 253 for normal values. The mantissas, including the leading bit of 1, are 24 bits long. Therefore, the output of the SHIFTER can be non-zero for  $n \leq 23$  only, ignoring guard bit considerations. Let  $n = n_7n_6 \dots n_0$ , and define an enable signal, EN, as  $EN = \bar{n}_7\bar{n}_6\bar{n}_5$ . This variable must be 1 for any output of the SHIFTER to be non-zero. Let  $m = m_{23}m_{22} \dots m_0$  and  $s_{23}s_{22} \dots s_0$  be the SHIFTER inputs and outputs, respectively. The largest network is required for output  $s_0$ , because any of the 24 input bits could be shifted into this output position. Define an intermediate binary vector  $i = i_{23}i_{22} \dots i_0$ . We will first shift  $m$  into  $i$  based on EN and  $n_4n_3$ . (Then we will shift  $i$  into  $s$ , based on  $n_2n_1n_0$ .) Only the part of  $i$  needed to generate  $s_0$  will be specified.

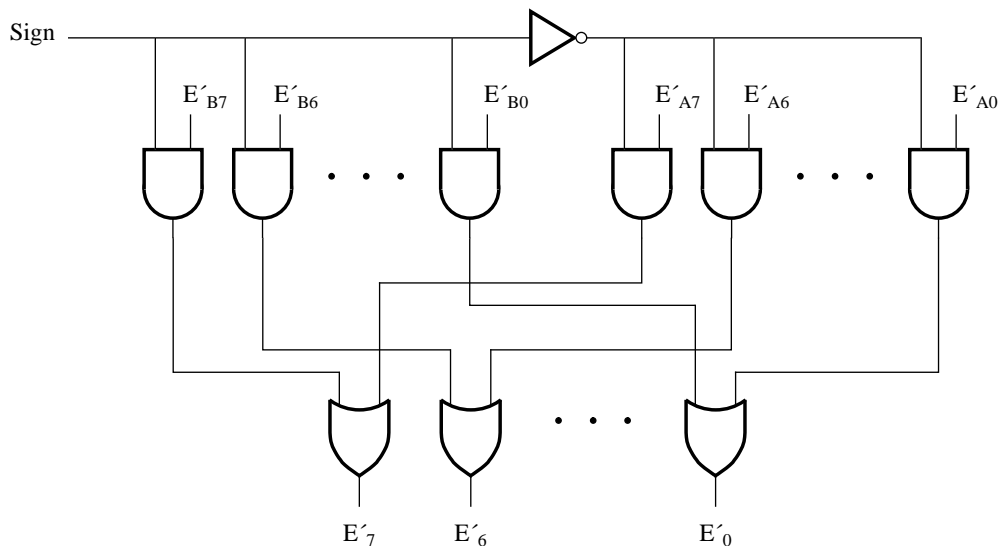
$$\begin{aligned}
i_7 &= ENn_4\bar{n}_3m_{23} + EN\bar{n}_4n_3m_{15} + EN\bar{n}_4\bar{n}_3m_7 \\
i_6 &= (\dots)m_{22} + (\dots)m_{14} + (\dots)m_6 \\
i_5 &= (\dots)m_{21} + (\dots)m_{13} + (\dots)m_5 \\
&\vdots \\
&\vdots \\
&\vdots \\
i_0 &= (\dots)m_{16} + (\dots)m_8 + (\dots)m_0
\end{aligned}$$

Gates with fan-in up to only 4 are needed to generate these 8 signals. Note that all bits of  $m$  are involved, as claimed. We now generate  $s_0$  from these signals and  $n_2n_1n_0$  as follows:

$$\begin{aligned}
s_0 &= n_2n_1n_0i_7 + n_2n_1\bar{n}_0i_6 + n_2\bar{n}_1n_0i_5 + n_2\bar{n}_1\bar{n}_0i_4 \\
&\quad + \bar{n}_2n_1n_0i_3 + \bar{n}_2n_1\bar{n}_0i_2 + \bar{n}_2\bar{n}_1n_0i_1 + \bar{n}_2\bar{n}_1\bar{n}_0i_0
\end{aligned}$$

Note that this requires a fan-in of 8 in the OR gate, so that 3 gates will be needed. Other  $s_i$  positions can be generated in a similar way.

6.34. (a)



(b) The SWAP network is a pair of multiplexers, each one similar to (a).

6.35. Let  $m = m_{24}m_{23} \dots m_0$  be the output of the adder/subtractor. The leftmost bit,  $m_{24}$ , is the overflow bit that could result from addition. (We ignore the handling of guard bits.) Derive a series of variables,  $z_i$ , as follows:

$$\begin{aligned}
 z_{-1} &= m_{24} \\
 z_0 &= \overline{m}_{24}m_{23} \\
 z_1 &= \overline{m}_{24}\overline{m}_{23}m_{22} \\
 &\vdots \\
 z_{23} &= \overline{m}_{24}\overline{m}_{23} \dots m_0 \\
 z_{24} &= \overline{m}_{24}\overline{m}_{23} \dots \overline{m}_0
 \end{aligned}$$

Note that exactly one of the  $z_i$  variables is equal to 1 for any particular  $m$  vector. Then encode these  $z_i$  variables, for  $-1 \leq i \leq 23$ , into a 6-bit signal representation for  $X$ , so that if  $z_i = 1$ , then  $X = i$ . The variable  $z_{24}$  signifies whether or not the resulting mantissa is zero.

- 6.36. Augment the 24-bit operand magnitudes entering the adder/subtractor by adding a sign bit position at the left end. Subtraction is then achieved by complementing the bottom operand and performing addition. Group corresponding bit-pairs from the two, signed, 25-bit operands into six groups of four bit-pairs each, plus one bit-pair at the left end, for purposes of deriving  $P_i$  and  $G_i$  functions. Label these functions  $P_6, G_6, \dots, P_0, G_0$ , from left-to-right, following the pattern developed in Section 6.2.

The lookahead logic must generate the group input carries  $c_0, c_4, c_8, \dots, c_{24}$ , accounting properly for the “end-around carry”. The key fact is that a carry  $c_i$  may have the value 1 because of a generate condition (i.e., some  $G_i = 1$ ) in a higher-order group as well as in a lower-order group. This observation leads to the following logic expressions for the carries:

$$\begin{aligned} c_0 &= G_6 + P_6G_5 + \dots + P_6P_5P_4P_3P_2P_1G_0 \\ c_4 &= G_0 + P_0G_6 + P_0P_6G_5 + \dots + P_0P_6P_5P_4P_3P_2G_1 \\ &\cdot \\ &\cdot \\ &\cdot \end{aligned}$$

Since the output of this adder is in 1’s-complement form, the sign bit determines whether or not to complement the remaining bits in order to send the magnitude  $M$  on to the “Normalize and Round” operation. Addition of positive numbers leading to overflow is a valid result, as discussed in Section 6.7.4, and must be distinguished from a negative result that may occur when subtraction is performed. Some logic at the left-end sign position solves this problem.

# Chapter 7 – Basic Processing Unit

- 7.1. The WMFC step is needed to synchronize the operation of the processor and the main memory.
- 7.2. Data requested in step 1 are fetched during step 2 and loaded into MDR at the end of that clock cycle. Hence, the total time needed is 7 cycles.
- 7.3. Steps 2 and 5 will take 2 cycles each. Total time = 9 cycles.
- 7.4. The minimum time required for transferring data from one register to register Z is equal to the propagation delay + setup time  
 $= 0.3 + 2 + 0.2 = 2.5 \text{ ns}$ .
- 7.5. For the organization of Figure 7.1:
  - (a) 1.  $PC_{out}, MAR_{in}, \text{Read}, \text{Select4}, \text{Add}, Z_{in}$   
 2.  $Z_{out}, PC_{in}, Y_{in}, \text{WMFC}$   
 3.  $MDR_{out}, IR_{in}$   
 4.  $PC_{out}, MAR_{in}, \text{Read}, \text{Select4}, \text{Add}, Z_{in}$   
 5.  $Z_{out}, PC_{in}, Y_{in}$   
 6.  $R1_{out}, Y_{in}, \text{WMFC}$   
 7.  $MDR_{out}, \text{SelectY}, \text{Add}, Z_{in}$   
 8.  $Z_{out}, R1_{in}, \text{End}$
  - (b) 1-4. Same as in (a)  
 5.  $Z_{out}, PC_{in}, \text{WMFC}$   
 6.  $MDR_{out}, MAR_{in}, \text{Read}$   
 7.  $R1_{out}, Y_{in}, \text{WMFC}$   
 8.  $MDR_{out}, \text{Add}, Z_{in}$   
 9.  $Z_{out}, R1_{in}, \text{End}$
  - (c) 1-5. Same as in (b)  
 6.  $MDR_{out}, MAR_{in}, \text{Read}, \text{WMFC}$   
 7-10. Same as 6-9 in (b)
- 7.6. Many approaches are possible. For example, the three machine instructions implemented by the control sequences in parts *a*, *b*, and *c* can be thought of as one instruction, Add, that has three addressing modes, Immediate (Imm), Absolute (Abs), and Indirect (Ind), respectively. In order to simplify the decoder block, hardware may be added to enable the control step counter to be conditionally loaded with an out-of-sequence number at any time. This provides a "branching" facility in the control sequence. The three control sequences may now be merged into one, as follows:
  - 1-4. Same as in (a)
  5.  $Z_{out}, PC_{in}, \text{If Imm branch to 10}$

6. WMFC
7.  $MDR_{out}$ ,  $MAR_{in}$ , Read, If Abs branch to 10
8. WMFC
9.  $MDR_{out}$ ,  $MAR_{in}$ , Read
10.  $R1_{out}$ ,  $Y_{in}$ , WMFC
11.  $MDR_{out}$ , Add,  $Z_{in}$
12.  $Z_{out}$ ,  $R1_{in}$ , End

Depending on the details of hardware timing, steps, 6 and 7 may be combined. Similarly, steps 8 and 9 may be combined.

- 7.7. Following the timing model of Figure 7.5, steps 2 and 5 take 16 ns each. Hence, the 7-step sequence takes 42 ns to complete, and the processor is idle  $28/42 = 67\%$  of the time.

- 7.8. Use a 4-input multiplexer with the inputs 1, 2, 4, and Y.

- 7.9. With reference to Figure 6.7, the control sequence needs to generate the Shift right and Add/Noadd (multiplexer control) signals and control the number of additions/subtractions performed. Assume that the hardware is configured such that register Z can perform the function of the accumulator, register TEMP can be used to hold the multiplier and is connected to register Z for shifting as shown. Register Y will be used to hold the multiplicand. Furthermore, the multiplexer at the input of the ALU has three inputs, 0, 4, and Y. To simplify counting, a counter register is available on the bus. It is decremented by a control signal Decrement and it sets an output signal Zero to 1 when it contains zero. A facility to place a constant value on the bus is also available.

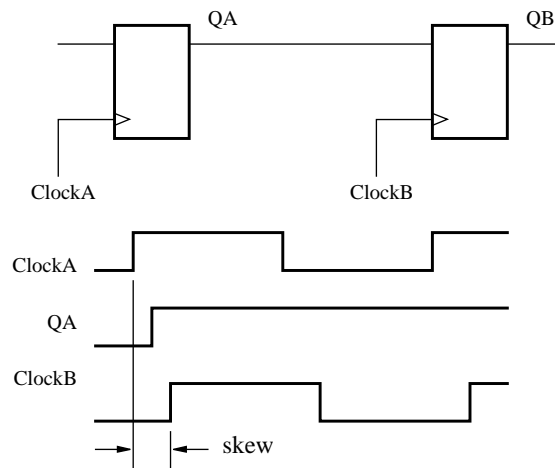
After fetching the instruction the control sequence continues as follows:

4. Constant=32,  $Constant_{out}$ ,  $Counter_{in}$
5.  $R1_{out}$ ,  $TEMP_{in}$
6.  $R2_{out}$ ,  $Y_{in}$
7.  $Z_{out}$ , if  $TEMP_0 = 1$  then SelectY else Select0, Add,  $Z_{in}$ , Decrement
8. Shift, if Zero=0 then Branch 7
9.  $Z_{out}$ ,  $R2_{in}$ , End

- 7.10. The control steps are:

- 1-3. Fetch instruction (as in Figure 7.9)
4.  $PC_{out}$ , Offset-field-of- $IR_{out}$ , Add, If N = 1 then  $PC_{in}$ , End

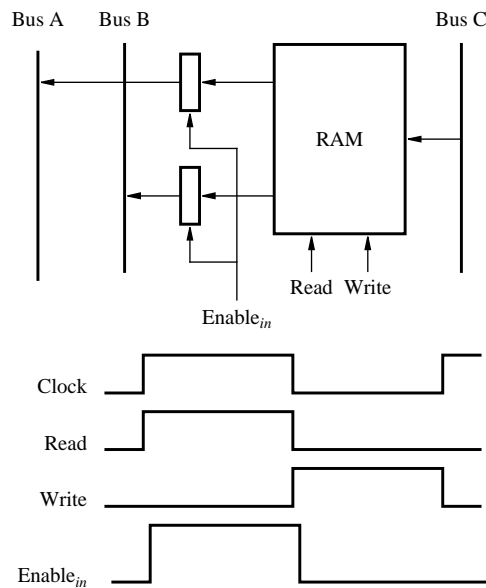
- 7.11. Let SP be the stack pointer register. The following sequence is for a processor that stores the return address on a stack in the memory.
- 1-3. Fetch instruction (as in Figure 7.6)
  4.  $SP_{out}$ , Select4, Subtract,  $Z_{in}$
  5.  $Z_{out}$ ,  $SP_{in}$ ,  $MAR_{in}$
  6.  $PC_{out}$ ,  $MDR_{in}$ , Write,  $Y_{in}$
  7. Offset-field-of- $IR_{out}$ , Add,  $Z_{in}$
  8.  $Z_{out}$ ,  $PC_{in}$ , End, WMFC
- 7.12. 1-3. Fetch instruction (as in Figure 7.9)
4.  $SP_{outB}$ , Select4, Subtract,  $SP_{in}$ ,  $MAR_{in}$
  5.  $PC_{out}$ , R=B,  $MDR_{in}$ , Write
  6. Offset-field-of- $IR_{out}$ ,  $PC_{out}$ , Add,  $PC_{in}$ , WMFC, End
- 7.13. The latch in Figure A.27 cannot be used to implement a register that can be both the source and the destination of a data transfer operation. For example, it cannot be used to implement register Z in Figure 7.1. It may be used in other registers, provided that hold time requirements are met.
- 7.14. The presence of a gate at the clock input of a flip-flop introduces clock skew. This means that clock edges do not reach all flip-flops at the same time. For example, consider two flip-flops A and B, with output QA connected to input DB. A clock edge loads new data into A, and the next clock edge transfers these data to B. However, if clock B is delayed, the new data loaded into A may reach B before the clock and be loaded into B one clock period too early.



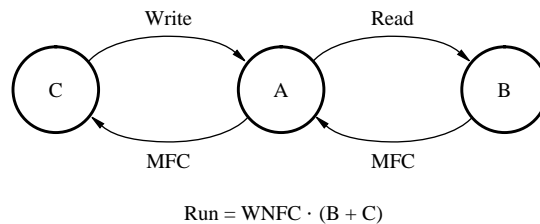
In the absence of clock skew, flip-flop B records a 0 at the first clock edge. However, if Clock B is delayed as shown, the flip-flop records a 1.



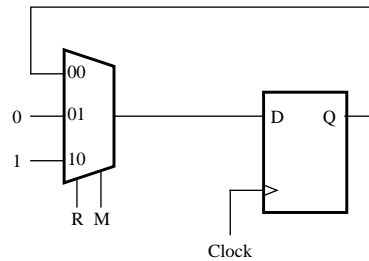
- 7.15. Add a latch similar to that in Figure A.27 at each of the two register file outputs. A read operation is performed in the RAM in the first half of a clock cycle and the latch inputs are enabled at that time. The data read enter the two latches and appear on the two buses immediately. During the second phase of the clock the latch inputs are disabled, locking the data in. Hence, the data read will continue to be available on the buses even if the outputs of the RAM change. The RAM performs a write operation during this phase to record the results of the data transfer.



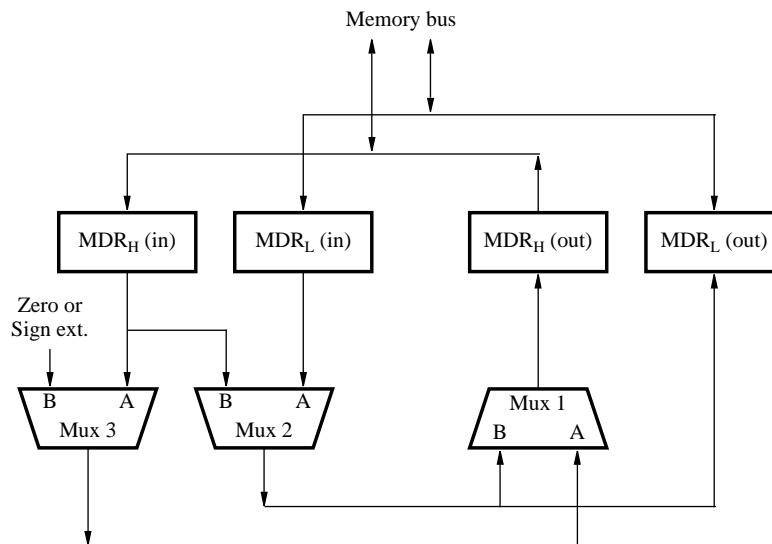
- 7.16. The step counter advances at the end of a clock period in which Run is equal to 1. With reference to Figure 7.5, Run should be set to 0 during the first clock cycle of step 2 and set to 1 as soon as MFC is received. In general, Run should be set to 0 by WMFC and returned to 1 when MFC is received. To account for the possibility that a memory operation may have been already completed by the time WMFC is issued, Run should be set to 0 only if the requested memory operation is still in progress. A state machine that controls bus operation and generates the run signal is given below.



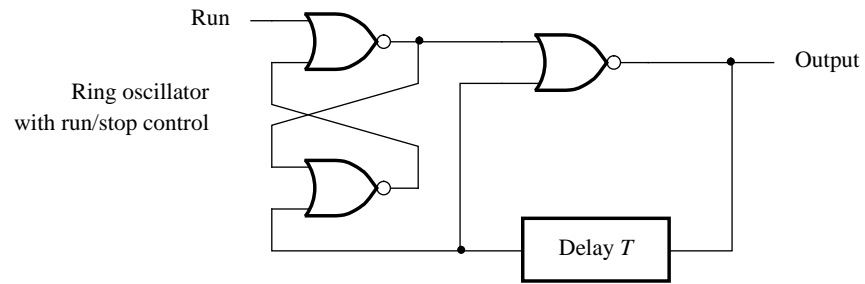
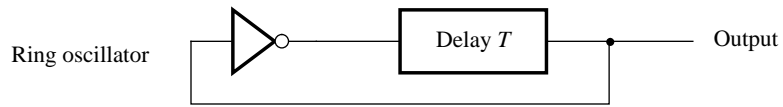
- 7.17. The following circuit uses a multiplexer arrangement similar to that in Figure 7.3.



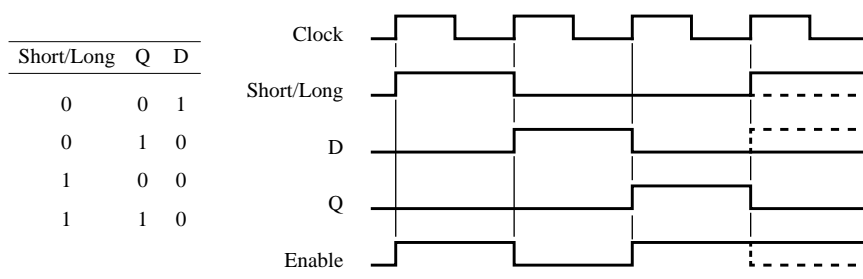
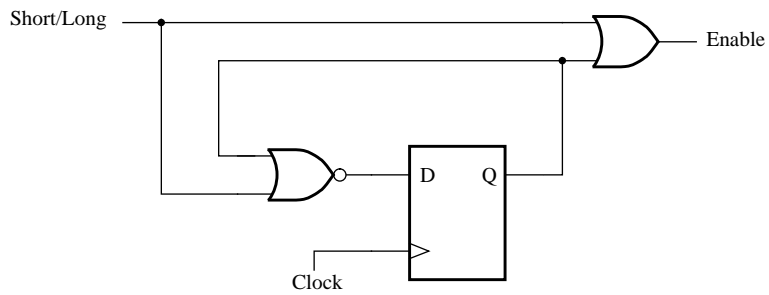
- 7.18. A possible arrangement is shown below. For clarity, we have assumed that MDR consists of two separate registers for input and output data. Multiplexers Mux-1 and Mux-2 select input B for even and input A for odd byte operations. Mux 3 selects input A for word operations and input B for byte operations. Input B provides either zero extension or sign extension of byte operands. For sign-extension it should be connected to the most-significant bit output of multiplexer Mux-2.



- 7.19. Use the delay element in a ring oscillator as shown below. The frequency of oscillation is  $1/(2T)$ . By adding the control circuit shown, the oscillator will run only while Run is equal to 1. When stopped, its output A is equal to 0. The oscillator will always generate complete output pulses. If Run goes to 0 while A is 1, the latch will not change state until B goes to 1 at the end of the pulse.



- 7.20. In the circuit below, Enable is equal to 1 whenever Short/Long is equal to 1, indicating a short pulse. When this line changes to 0, Enable changes to 0 for one clock cycle.



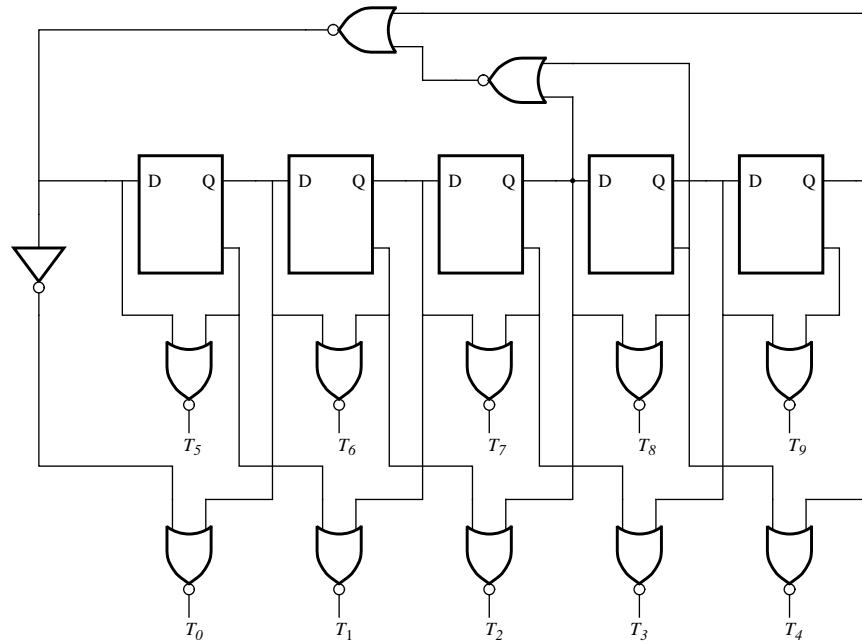
7.21. (a) Count sequence is: 0000 1000 1100 1110 1111 0111 0011 0001 0000

(b) A 5-bit Johnson counter is shown below, with the outputs  $Q_1$  to  $Q_5$  decoded to generate the signals  $T_1$  to  $T_{10}$ . The feed back circuit has been modified to make the counter self-starting. It implements the function

$$D_1 = \overline{Q_5 + Q_3 + \overline{Q_4}}$$

This circuit detects states that have  $Q_3Q_4Q_5 = 010$  and changes the feedback value from 1 to 0. Without this or a similar modification to the feedback circuit, the counter may be stuck in sequences other than the desired one above.

The advantage of a Johnson counter is that there are no glitches in decoding the count value to generate the timing signals.



7.22. We will generate a signal called Store to recirculate data when no external action is required.

$$\text{Store} = \overline{(\text{ARS} + \text{LSR} + \text{SL} + \text{LLD})}$$

$$D_{15} = \text{ASR} \cdot Q_{15} + \text{SL} \cdot Q_{14} + \text{ROR} \cdot \text{Carry} + \text{LD} \cdot D_{15} + \text{Store} \cdot Q_{15}$$

$$D_1 = (\text{ASR} + \text{LSR} + \text{ROR}) \cdot Q_2 + \text{SL} \cdot Q_0 + \text{LD} \cdot D_1 + \text{Store} \cdot Q_1$$

$$D_0 = (\text{ASR} + \text{LSR} + \text{ROR}) \cdot Q_1 + \text{LD} \cdot D_0 + \text{Store} \cdot Q_0$$

7.23. A state diagram for the required controller is given below. This is a Moore machine. The output values are given inside each state as they are functions of the state only.

Since there are 6 independent states, a minimum of three flip-flops  $r$ ,  $s$ , and  $t$  are required for the implementation. A possible state assignment is shown in the diagram. It has been chosen to simplify the generation of the outputs  $X$ ,  $Y$ , and  $Z$ , which are given by

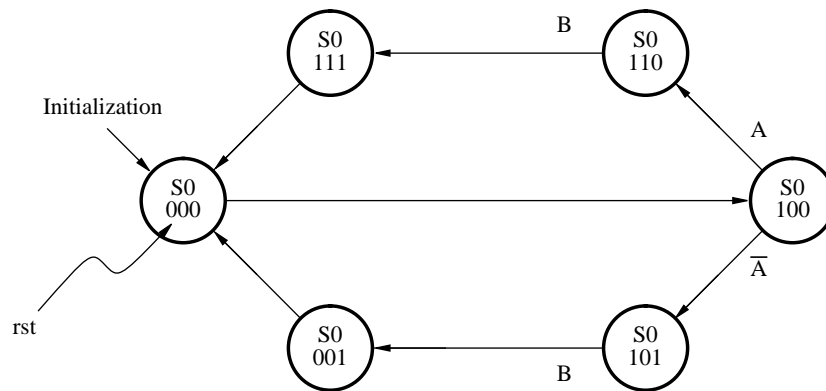
$$X = r + s + t \quad Y = s \quad Z = t$$

Using D flip-flops for implementation of the controller, the required inputs to the flip-flops may be generated as follows

$$D(r) = s \bar{t} B + \bar{s} \bar{t}$$

$$D(s) = \bar{s} \bar{t} A + s \bar{t} B$$

$$D(t) = s \bar{t} B + \bar{s} \bar{t} \bar{A} + \bar{s} t B$$



7.24. Microroutine:

| Address<br>(Octal) | Microinstruction                                                                                                     |
|--------------------|----------------------------------------------------------------------------------------------------------------------|
| 000-002            | Same as in Figure 7.21                                                                                               |
| 300                | $\mu\text{Branch } \{\mu\text{PC} \leftarrow 161\}$                                                                  |
| 161                | $\text{PC}_{out}, \text{MAR}_{in}, \text{Read}, \text{Select4}, \text{Add}, \text{Z}_{in}$                           |
| 162                | $\text{Z}_{out}, \text{PC}_{in}, \text{WMFC}$                                                                        |
| 163                | $\text{MDR}_{out}, \text{Y}_{in}$                                                                                    |
| 164                | $\text{Rsrc}_{out}, \text{SelectY}, \text{Add}, \text{Z}_{in}$                                                       |
| 165                | $\text{Z}_{out}, \text{MAR}_{in}, \text{Read}$                                                                       |
| 166                | $\mu\text{Branch } \{\mu\text{PC} \leftarrow 170; \mu\text{PC}_0 \leftarrow [\overline{\text{IR}_8}]\}, \text{WMFC}$ |
| 170-173            | Same as in Figure 7.21                                                                                               |

7.25. Conditional branch

| Address<br>(Octal) | Microinstruction                                                                                      |
|--------------------|-------------------------------------------------------------------------------------------------------|
| 000-002            | Same as in Figure 7.21                                                                                |
| 003                | $\mu\text{Branch } \{\mu\text{PC} \leftarrow 300\}$                                                   |
| 300                | if $\text{Z}+(\text{N} \oplus \text{V}) = 1$ then $\mu\text{Branch } \{\mu\text{PC} \leftarrow 304\}$ |
| 301                | $\text{PC}_{out}, \text{Y}_{in}$                                                                      |
| 302                | $\text{Address}_{out}, \text{SelectY}, \text{Add}, \text{Z}_{in}$                                     |
| 303                | $\text{Z}_{out}, \text{PC}_{in}, \text{End}$                                                          |

7.26. Assume microroutine starts at 300 for all three instructions. (Alternatively, the instruction decoder may branch to 302 directly in the case of an unconditional branch instruction.)

| Address<br>(Octal) | Microinstruction                                                                                      |
|--------------------|-------------------------------------------------------------------------------------------------------|
| 000-002            | Same as in Figure 7.21                                                                                |
| 003                | $\mu\text{Branch } \{\mu\text{PC} \leftarrow 300\}$                                                   |
| 300                | if $\text{Z}+(\text{N} \oplus \text{V}) = 1$ then $\mu\text{Branch } \{\mu\text{PC} \leftarrow 000\}$ |
| 301                | if $(\text{N} = 1)$ then $\mu\text{Branch } \{\mu\text{PC} \leftarrow 000\}$                          |
| 302                | $\text{PC}_{out}, \text{Y}_{in}$                                                                      |
| 303                | $\text{Offset-field-of-IR}_{out}, \text{SelectY}, \text{Add}, \text{Z}_{in}$                          |
| 304                | $\text{Z}_{out}, \text{PC}_{in}, \text{End}$                                                          |

- 7.27. The answer to problem 3.26 holds in this case as well, with the restriction that one of the operand locations (either source or destination) must be a data register.

| Address<br>(Octal) | Microinstruction                                                                             |
|--------------------|----------------------------------------------------------------------------------------------|
| 000-002            | Same as in Figure 7.21                                                                       |
| 003                | $\mu\text{Branch } \{\mu\text{PC} \leftarrow 010\}$                                          |
| 010                | if ( $\text{IR}_{10-8} = 000$ ) then $\mu\text{Branch } \{\mu\text{PC} \leftarrow 101\}$     |
| 011                | if ( $\text{IR}_{10-8} = 001$ ) then $\mu\text{Branch } \{\mu\text{PC} \leftarrow 111\}$     |
| 012                | if ( $\text{IR}_{10-9} = 01$ ) then $\mu\text{Branch } \{\mu\text{PC} \leftarrow 121\}$      |
| 013                | if ( $\text{IR}_{10-9} = 10$ ) then $\mu\text{Branch } \{\mu\text{PC} \leftarrow 141\}$      |
| 014                | $\mu\text{Branch } \{\mu\text{PC} \leftarrow 161\}$                                          |
| 121                | $\text{Rsrc}_{out}, \text{MAR}_{in}, \text{Read}, \text{Select4}, \text{Add}, \text{Z}_{in}$ |
| 122                | $\text{Z}_{out}, \text{Rsrc}_{in}$                                                           |
| 123                | if ( $\text{IR}_8 = 1$ ) then $\mu\text{Branch } \{\mu\text{PC} \leftarrow 171\}$            |
| 124                | $\mu\text{Branch } \{\mu\text{PC} \leftarrow 170\}$                                          |
| 170-173            | Same as in Figure 7.21                                                                       |

- 7.28. There is no change for the five address modes in Figure 7.20. Absolute and Immediate modes require a separate path. However, some sharing may be possible among absolute, immediate, and indexed, as all three modes read the word following the instruction. Also, Full Indexed mode needs to be implemented by adding the contents of the second register to generate the effective address. After each memory access, the program counter should be updated by 2, rather than 4, in the case of the 16-bit processor.
- 7.29. The same general structure can be used. Since the dst operand can be specified in any of the five addressing modes as the src operand, it is necessary to replicate the microinstructions that determine the effective address of an operand. At microinstruction 172, the source operand should be placed in a temporary register and another tree of microinstructions should be entered to fetch the destination operand.

7.30. (a) A possible address assignment is as follows.

| Address | Microinstruction                             |
|---------|----------------------------------------------|
| 0000    | A                                            |
| 0001    | B                                            |
| 0010    | if ( $b_6b_5$ ) = 00) then $\mu$ Branch 0111 |
| 0011    | if ( $b_6b_5$ ) = 01) then $\mu$ Branch 1010 |
| 0100    | if ( $b_6b_5$ ) = 10) then $\mu$ Branch 1100 |
| 0101    | I                                            |
| 0110    | $\mu$ Branch 1111                            |
| 0111    | C                                            |
| 1000    | D                                            |
| 1001    | $\mu$ Branch 1111                            |
| 1010    | E                                            |
| 1011    | $\mu$ Branch 1111                            |
| 1100    | F                                            |
| 1101    | G                                            |
| 1110    | H                                            |
| 1111    | J                                            |

(b) Assume that bits  $b_{6-5}$  of IR are ORed into bit  $\mu PC_{3-2}$

| Address | Microinstruction                     |
|---------|--------------------------------------|
| 0000    | A                                    |
| 0001    | B; $\mu PC_{3-2} \leftarrow b_{6-5}$ |
| 0010    | C                                    |
| 0011    | D                                    |
| 0100    | $\mu$ Branch 1111                    |
| 0101    | E                                    |
| 0110    | $\mu$ Branch 1111                    |
| 0111    | F                                    |
| 1011    | G                                    |
| 1100    | H                                    |
| 1101    | $\mu$ Branch 1111                    |
| 1110    | I                                    |
| 1111    | J                                    |



(c)

| Address | Microinstruction |                                      |
|---------|------------------|--------------------------------------|
|         | Next address     | Function                             |
| 0000    | 0001             | A                                    |
| 0001    | 0010             | B; $\mu PC_{3-2} \leftarrow b_{6-5}$ |
| 0010    | 0011             | C                                    |
| 0011    | 1111             | D                                    |
| 0110    | 1111             | E                                    |
| 1010    | 1011             | F                                    |
| 1011    | 1100             | G                                    |
| 1100    | 1111             | H                                    |
| 1110    | 1111             | I                                    |
| 1111    | —                | J                                    |

- 7.31. Put the  $Y_{in}$  control signal as the fourth signal in F5, to reduce F3 by one bit. Combine fields F6, F7, and F8 into a single 2-bit field that represents:

00: Select4  
01: SelectY  
10: WMFC  
11: End

Combining signals means that they cannot be issued in the same microinstruction.

- 7.32. To reduce the number of bits, we should use larger fields that specify more signals. This, inevitably, leads to fewer choices in what signals can be activated at the same time. The choice as to which signals can be combined should take into account what signals are likely to be needed in a given step.

One way to provide flexibility is to define control signals that perform multiple functions. For example, whenever MAR is loaded, it is likely that a read command should be issued. We can use two signals:  $MAR_{in}$  and  $MAR_{in} \cdot \text{Read}$ . We activate the second one when a read command is to be issued. Similarly,  $Z_{in}$  is always accompanied by either Select Y or Select4. Hence, instead of these three signals, we can use  $Z_{in} \cdot \text{Select4}$  and  $Z_{in} \cdot \text{SelectY}$ .

A possible 12-bit encoding uses three 4-bit fields FA, FB, and FC, which combine signals from Figure 7.19 as follows:

FA: F1 plus,  $Z_{out} \cdot \text{End}$ ,  $Z_{out} \cdot \text{WMFC}$ . (11 signals)

FB: F2, F3, Instead of  $Z_{in}$ ,  $MAR_{in}$ , and  $MDR_{in}$  use  $Z_{in} \cdot \text{Select4}$ ,  $Z_{in} \cdot \text{SelectY}$ ,  $MAR_{in}$ ,  $MAR_{in} \cdot \text{Read}$ , and  $MDR_{in} \cdot \text{Write}$ . (13 signals)

FC: F4 (16 signals)

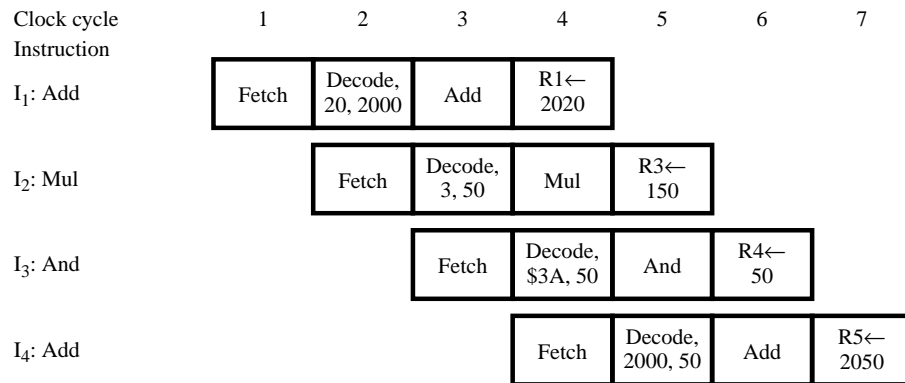
With these choices, step 5 in Figure 7.6 must be split into two steps, leading to an 8-step sequence. Figure 7.7 remains unchanged.

- 7.33. Figure 7.8 contains two buses, A and B, one connected to each of the two inputs of the ALU. Therefore, two fields are needed instead of F1; one field to provide gating of registers onto bus A, and another onto bus B.
- 7.34. Horizontal microinstructions are longer. Hence, they require a larger microprogram memory. A vertical organization requires more encoding and decoding of signals, hence longer delays, and leads to longer microprograms and slower operation. With the high-density of today's integrated circuits, the vertical organization is no longer justified.
- 7.35. The main advantage of hardwired control is fast operation. The disadvantages include: higher cost, inflexibility when changes or additions are to be made, and longer time required to design and implement such units.

Microprogrammed control is characterized by low cost and high flexibility. Lower speed of operation becomes a problem in high-performance computers.

## Chapter 8 – Pipelining

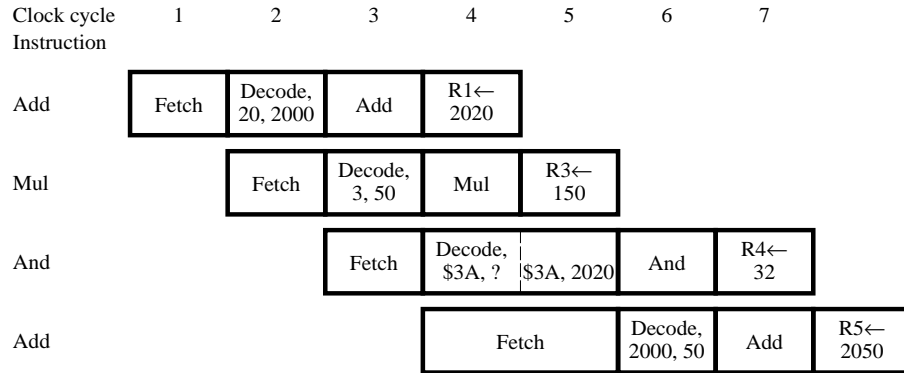
8.1. (a) The operation performed in each step and the operands involved are as given in the figure below.



(b)

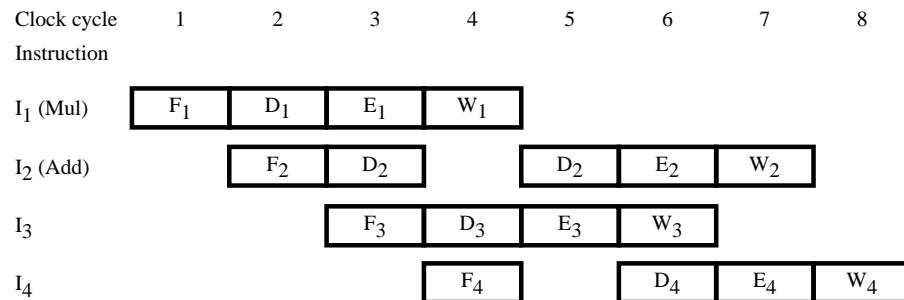
| Clock cycle | 2                                       | 3                                                      | 4                                                      | 5                                                      |
|-------------|-----------------------------------------|--------------------------------------------------------|--------------------------------------------------------|--------------------------------------------------------|
| Buffer B1   | Add instruction (I <sub>1</sub> )       | Mul instruction (I <sub>2</sub> )                      | And instruction (I <sub>3</sub> )                      | Add instruction (I <sub>4</sub> )                      |
| Buffer B2   | Information from a previous instruction | Decoded I <sub>1</sub><br>Source operands:<br>20, 2000 | Decoded I <sub>2</sub><br>Source operands:<br>3, 50    | Decoded I <sub>3</sub><br>Source operands:<br>\$3A, 50 |
| Buffer B3   | Information from a previous instruction | Information from a previous instruction                | Result of I <sub>1</sub> :<br>2020<br>Destination = R1 | Result of I <sub>2</sub> :<br>150<br>Destination = R3  |

8.2. (a)

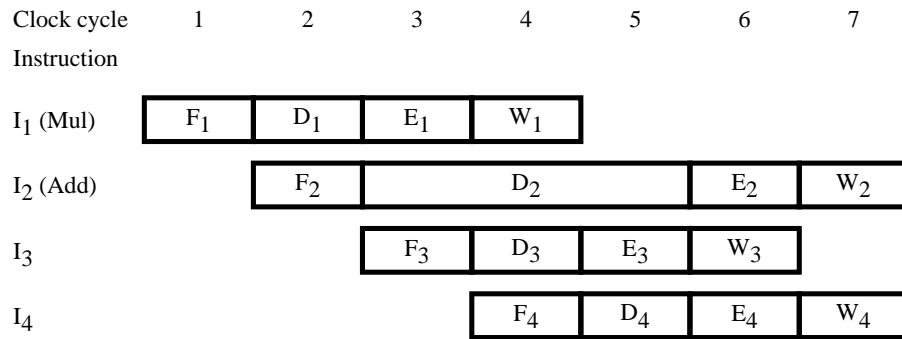


(b) Cycles 2 to 4 are the same as in P8.1, but contents of R1 are not available until cycle 5. In cycle 5, B1 and B2 have the same contents as in cycle 4. B3 contains the result of the multiply instruction.

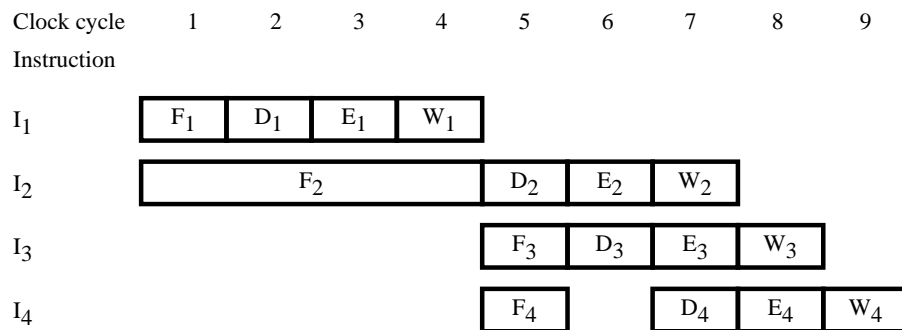
8.3. Step D<sub>2</sub> may be abandoned, to be repeated in cycle 5, as shown below. But, instruction I<sub>1</sub> must remain in buffer B1. For I<sub>3</sub> to proceed, buffer B1 must be capable of holding two instructions. The decode step for I<sub>4</sub> has to be delayed as shown, assuming that only one instruction can be decoded at a time.



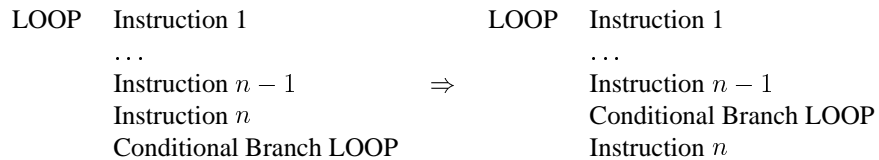
8.4. If all decode and execute stages can handle two instructions at a time, only instruction  $I_2$  is delayed, as shown below. In this case, all buffers must be capable of holding information for two instructions. Note that completing instruction  $I_3$  before  $I_2$  could cause problems. See Section 8.6.1.



8.5. Execution proceeds as follows.



8.6. The instruction immediately preceding the branch should be placed after the branch.



This reorganization is possible only if the branch instruction does not depend on instruction  $n$ .

- 8.7. The UltraSPARC arrangement is advantageous when the branch instruction is at the end of the loop and it is possible to move one instruction from the body of the loop into the delay slot. The alternative arrangement is advantageous when the branch instruction is at the beginning of the loop.
- 8.8. The instruction executed on a speculative basis should be one that is likely to be the correct choice most often. Thus, the conditional branch should be placed at the end of the loop, with an instruction from the body of the loop moved to the delay slot if possible. Alternatively, a copy of the first instruction in the loop body can be placed in the delay slot and the branch address changed to that of the second instruction in the loop.
- 8.9. The first branch (BLE) has to be followed by a NOP instruction in the delay slot, because none of the instructions around it can be moved. The inner and outer loop controls can be adjusted as shown below. The first instruction in the outer loop is duplicated in the delay slot following BLE. It will be executed one more time than in the original program, changing the value left in R3. However, this should cause no difficulty provided the contents of R3 are not needed once the sort is completed. The modified program is as follows:

|       |          |            |                    |
|-------|----------|------------|--------------------|
|       | ADD      | R0,LIST,R3 |                    |
|       | ADD      | R0,N,R1    |                    |
|       | SUB      | R1,1,R1    |                    |
|       | SUB      | R1,1,R2    |                    |
| OUTER | LDUB     | [R3+R1],R5 | Get LIST(j)        |
|       | LDUB     | [R3+R2],R6 | Get LIST(k)        |
| INNER | SUB      | R6,R5,R0   |                    |
|       | BLE,pt   | NEXT       |                    |
|       | SUB      | R2,1,R2    | $k \leftarrow k-1$ |
|       | STUB     | R5,[R3+R2] |                    |
|       | STUB     | R6,[R3+R1] |                    |
|       | OR       | R0,R6,R5   |                    |
| NEXT  | BGE,pt,a | INNER      |                    |
|       | LDUB     | [R3+R2],R6 | Get LIST(k)        |
|       | SUB      | R1,1,R1    |                    |
|       | BGT,pt   | OUTER      |                    |
|       | SUB      | R1,1,R2    |                    |

8.10. Without conditional instructions:

|         |          |         |                          |
|---------|----------|---------|--------------------------|
|         | Compare  | A,B     | Check A – B              |
|         | Branch>0 | Action1 |                          |
| Action2 | ...      | ...     | One or more instructions |
|         | Branch   | Next    |                          |
| Action1 | ...      | ...     | One or more instructions |
| Next    | ...      |         |                          |

If conditional instructions are available, we can use:

|      |         |     |                                     |
|------|---------|-----|-------------------------------------|
|      | Compare | A,B | Check A – B                         |
|      | ...     | ... | Action1 instruction(s), conditional |
|      | ...     | ... | Action2 instruction(s), conditional |
| Next | ...     |     |                                     |

In the second case, all Action 1 and Action 2 instructions must be fetched and decoded to determine whether they are to be executed. Hence, this approach is beneficial only if each action consists of one or two instructions.


#### Without conditional instructions

|                    |                |                |                |                |                |                |
|--------------------|----------------|----------------|----------------|----------------|----------------|----------------|
| <b>Clock cycle</b> | 1              | 2              | 3              | 4              | 5              | 6              |
| <b>Instruction</b> |                |                |                |                |                |                |
| Compare A,B        | F <sub>1</sub> | E <sub>1</sub> |                |                |                |                |
| Branch>0 Action1   |                | F <sub>2</sub> | E <sub>2</sub> |                |                |                |
| Action2 ...        |                |                | F <sub>3</sub> | E <sub>3</sub> |                |                |
| Branch Next        |                |                |                | F <sub>4</sub> | E <sub>4</sub> |                |
| Action1 ...        |                |                |                |                |                |                |
| Next ...           |                |                |                |                | F <sub>6</sub> | E <sub>1</sub> |

#### With conditional instructions

|                    |                |                |                |                |                |  |
|--------------------|----------------|----------------|----------------|----------------|----------------|--|
| Compare A,B        | F <sub>1</sub> | E <sub>1</sub> |                |                |                |  |
| If >0 then action1 |                | F <sub>2</sub> | E <sub>2</sub> |                |                |  |
| If ≤0 then action2 |                |                | F <sub>3</sub> | E <sub>3</sub> |                |  |
| NEXT ...           |                |                |                | F <sub>4</sub> | E <sub>4</sub> |  |

8.11. Buffer contents will be as shown below.

|               |                                                                                    |       |                |  |
|---------------|------------------------------------------------------------------------------------|-------|----------------|--|
| Clock         |  |       |                |  |
| Cycle No.     | 3                                                                                  | 4     | 5              |  |
| ALU Operation | +                                                                                  | Shift | O <sub>3</sub> |  |
| R3            | 45                                                                                 | 130   | 260            |  |
| RSLT          | 198                                                                                | 130   | 260            |  |

8.12. Using Load and Store instructions, the program may be revised as follows:

|           |          |                     |
|-----------|----------|---------------------|
| INSERTION | Test     | RHEAD               |
|           | Branch>0 | HEAD                |
|           | Move     | RNEWREC,RHEAD       |
|           | Return   |                     |
| HEAD      | Load     | RTEMP1,(RHEAD)      |
|           | Load     | RTEMP2,(RNEWREC)    |
|           | Compare  | RTEMP1,RTEMP2       |
|           | Branch>0 | SEARCH              |
|           | Store    | RHEAD,4(RNEWREC)    |
|           | Move     | RNEWREC,RHEAD       |
|           | Return   |                     |
| SEARCH    | Move     | RHEAD,RCURRENT      |
| LOOP      | Load     | RNEXT,4(RCURRENT)   |
|           | Test     | RNEXT               |
|           | Branch=0 | TAIL                |
|           | Load     | RTEMP1,(RNEXT)      |
|           | Load     | RTEMP2,(RNEWREC)    |
|           | Compare  | RTEMP1,RTEMP2       |
|           | Branch<0 | INSERT              |
|           | Move     | RNEXT,RCURRENT      |
|           | Branch   | LOOP                |
| INSERT    | Store    | RNEXT,4(RNEWREC)    |
| TAIL      | Store    | RNEWREC,4(RCURRENT) |
|           | Return   |                     |

This program contains many dependencies and branch instructions. There very few possibilities for instruction reordering. The critical part where optimization should be attempted is the loop. Given that no information is available on branch behavior or delay slots, the only optimization possible is to separate instructions that depend on each. This would reduce the probability of stalling the pipeline.

The loop may be reorganized as follows.



```

LOOP    Load    RNEXT,4(RCURRENT)
        Load    RTEMP2,(RNEWREC)
        Test    RNEXT
        Load    RTEMP1,(RNEXT)
        Branch=0 TAIL
        Compare RTEMP1,RTEMP2
        Branch<0 INSERT
        Move    RNEXT,RCURRENT
        Branch  LOOP
INSERT  Store    RNEXT,4(RNEWREC)
TAIL    Store    RNEWREC,4(RCURRENT)
        Return

```

Note that we have assumed that the Load instruction does not affect the condition code flags.

- 8.13. Because of branch instructions, 120 clock cycles are needed to execute 100 program instructions when delay slots are not used. Using the delay slots will eliminate 0.85 of the idle cycles. Thus, the improvement is given by:

$$\frac{120}{120 - 20 \times 0.85} = 1.081$$

That is, instruction throughput will increase by 8.1%.

- 8.14. Number of cycles needed to execute 100 instructions:

|                                                              |     |
|--------------------------------------------------------------|-----|
| Without optimization                                         | 140 |
| With optimization ( $140 - 20 \times 0.85 - 20 \times 0.2$ ) | 127 |

Thus, throughput improvement is  $140/127 = 1.102$ , or 10.2%

- 8.15. Throughput improvement due to pipelining is  $n$ , where  $n$  is the number of pipeline stages.

|          | Number of cycles needed to execute one instruction: | Throughput      |
|----------|-----------------------------------------------------|-----------------|
| 4-stage: | $1.20 - 0.8 \times 0.20 = 1.04$                     | $4/1.04 = 3.85$ |
| 6-stage: | $1.4 - 0.8 \times 0.2 - 0.25 \times 0.2 = 1.19$     | $6/1.19 = 5.04$ |

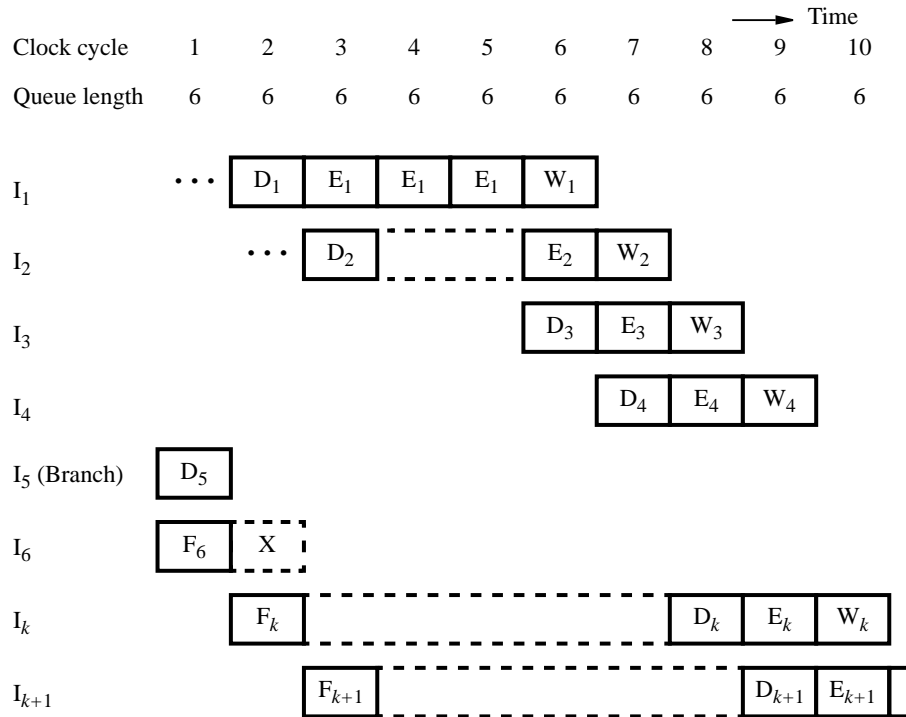
Thus, the 6-stage pipeline leads to higher performance.

8.16. For a “do while” loop, the termination condition is tested at the beginning of the loop. A conditional branch at that location will be taken when exiting the loop. Hence, it should be predicted not taken. That is, the state machine should be started in the state LNT, unless the loop is not likely to be executed at all.

A “do until” loop is executed at least once, and the branch condition is tested at the end of the loop. Assuming that the loop is likely to be executed several times, the branch should be predicted taken. That is, the state machine should be started in state LT.

8.17. An instruction fetched in cycle  $j$  reaches the head of the queue and enters the decode stage in cycle  $j + 6$ . Assume that the instruction preceding  $I_1$  is decoded and instruction  $I_6$  is fetched in cycle 1. This leads to instructions  $I_1$  to  $I_6$  being in the queue at the beginning of cycle 2. Execution would then proceed as shown below.

Note that the queue is always full, because at most one instruction is dispatched and up to two instructions are fetched in any given cycle. Under these conditions, the queue length would drop below 6 only in the case of a cache miss.



## Chapter 9 – Embedded Systems

- 9.1. Connect character input to the serial port and the 7-segment display unit to parallel port A. Connect bits  $PAOUT_6$  to  $PAOUT_0$  to the display segments  $a$  to  $g$ , respectively. Use the segment encoding shown in Figure A.37. For example, the decimal digit 0 sets the segments  $a, b, \dots, g$  to the hex pattern 7E.

A suitable program may use a table to convert the ASCII characters into the hex patterns for the display. The ASCII-encoded digits (see Table E.2) are represented by the pattern 111 in bit positions  $b_{6-4}$  and the corresponding BCD value (see Table E.1) in bit positions  $b_{3-0}$ . Hence, extracting the bits  $b_{3-0}$  from the ASCII code provides an index,  $j$ , which can be used to access the required entry in the conversion table (list). A possible program is obtained by modifying the program in Figure 9.11 as follows:

```
/* Define register addresses */
#define RBUF (volatile char *) 0xFFFFFEE0
#define SSTAT (volatile char *) 0xFFFFFEE2
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2

void main()
{
    /* Define the conversion table */
    char seg7[10] = {0x7E, 0x30, 0x6C, 0x79, 0x33, 0x5B, 0x5F, 0x30, 0x3F, 0x3B};
    char j;
    /* Initialize the parallel port */
    *PADIR = 0xFF; /* Configure Port A as output */

    /* Transfer the characters */
    while (1) { /* Infinite loop */
        while ((*SSTAT & 0x1) == 0); /* Wait for a new character */
        j = *RBUF & 0xF; /* Extract the BCD value */
        *PAOUT = seg7[j]; /* Send the 7-segment code to Port A */
    }
}
```

- 9.2. The arrangement explained in the solution for Problem 9.1 can be used. The entries in the conversion table can be accessed using the indexed addressing mode. Let the table occupy ten bytes starting at address SEG7. Then, using register R0 as the index register, the table is accessed using the mode SEG7(R0). The desired program may be obtained by modifying the program in Figure 9.10 as follows:

|                               |          |                                                            |                                       |
|-------------------------------|----------|------------------------------------------------------------|---------------------------------------|
| RBUF                          | EQU      | \$FFFFFFE0                                                 | Receive buffer.                       |
| SSTAT                         | EQU      | \$FFFFFFE2                                                 | Status register for serial interface. |
| PAOUT                         | EQU      | \$FFFFFFF1                                                 | Port A output data.                   |
| PADIR                         | EQU      | \$FFFFFFF2                                                 | Port A direction register.            |
| * Define the conversion table |          |                                                            |                                       |
|                               | ORIGIN   | \$200                                                      |                                       |
| SEG7                          | DataByte | \$7E, \$30, \$6C, \$79, \$33, \$5B, \$5F, \$30, \$3F, \$3B |                                       |
| * Initialization              |          |                                                            |                                       |
|                               | ORIGIN   | \$1000                                                     |                                       |
|                               | MoveByte | #\$FF,PADIR                                                | Configure Port A as output.           |
| * Transfer the characters     |          |                                                            |                                       |
| LOOP                          | Testbit  | #0,SSTAT                                                   | Check if new character is ready.      |
|                               | Branch=0 | LOOP                                                       |                                       |
|                               | MoveByte | RBUF,R0                                                    | Transfer a character to R0.           |
|                               | And      | #\$F,R0                                                    | Extract the BCD value.                |
|                               | MoveByte | SEG7(R0),PAOUT                                             | Send the 7-segment code to Port A.    |
|                               | Branch   | LOOP                                                       |                                       |

9.3. The arrangement explained in the solution for Problem 9.1 can be used. The desired program may be obtained by modifying the program in Figure 9.16 as follows:

```

/* Define register addresses */
#define RBUF (volatile char *) 0xFFFFFE0
#define SCNT (char *) 0xFFFFFE3
#define PAOUT (char *) 0xFFFFF1
#define PADIR (char *) 0xFFFFF2
#define int_addr (int *) (0x24)

void intserv();

void main()
{
    /* Define the conversion table */
    char seg7[10] = {0x7E, 0x30, 0x6C, 0x79, 0x33, 0x5B, 0x5F, 0x30, 0x3F, 0x3B};
    char j;
    /* Initialize the parallel port */
    *PADIR = 0xFF;          /* Configure Port A as output */

    /* Initialize the interrupt mechanism */
    int_addr = &intserv;    /* Set interrupt vector */
    __asm__("Move #0x40,%PSR"); /* Processor responds to IRQ interrupts */
    *SCNT = 0x10;          /* Enable receiver interrupts */

    /* Transfer the characters */
    while (1);              /* Infinite loop */
}

/* Interrupt service routine */
void intserv()
{
    j = *RBUF & 0xF;        /* Extract the BCD value */
    *PAOUT = seg7[j];        /* Send the 7-segment code to Port A */
    __asm__("ReturnI");      /* Return from interrupt */
}

```

- 9.4. The arrangement explained in the solutions for Problems 9.1 and 9.2 can be used. The desired program may be obtained by modifying the program in Figure 9.14 as follows:

|                               |          |                                                            |                                        |
|-------------------------------|----------|------------------------------------------------------------|----------------------------------------|
| RBUF                          | EQU      | \$FFFFFFE0                                                 | Receive buffer.                        |
| SCONT                         | EQU      | \$FFFFFFE3                                                 | Control register for serial interface. |
| PAOUT                         | EQU      | \$FFFFFFF1                                                 | Port A output data.                    |
| PADIR                         | EQU      | \$FFFFFFF2                                                 | Port A direction register.             |
| * Define the conversion table |          |                                                            |                                        |
|                               | ORIGIN   | \$200                                                      |                                        |
| SEG7                          | DataByte | \$7E, \$30, \$6C, \$79, \$33, \$5B, \$5F, \$30, \$3F, \$3B |                                        |
| * Initialization              |          |                                                            |                                        |
|                               | ORIGIN   | \$1000                                                     |                                        |
|                               | MoveByte | #\$FF,PADIR                                                | Configure Port A as output.            |
|                               | Move     | #INTSERV,\$24                                              | Set the interrupt vector.              |
|                               | Move     | #\$40,PSR                                                  | Processor responds to IRQ interrupts.  |
|                               | MoveByte | #\$10,SCONT                                                | Enable receiver interrupts.            |
| * Transfer loop               |          |                                                            |                                        |
| LOOP                          | Branch   | LOOP                                                       | Infinite wait loop.                    |
| * Interrupt service routine   |          |                                                            |                                        |
| INTSERV                       | MoveByte | RBUF,R0                                                    | Transfer a character to R0.            |
|                               | And      | #\$F,R0                                                    | Extract the BCD value.                 |
|                               | MoveByte | SEG7(R0),PAOUT                                             | Send the 7-segment code to Port A.     |
|                               | ReturnI  |                                                            | Return from interrupt.                 |

9.5. The arrangement explained in the solution for Problem 9.1 can be used, having the 7-segment displays connected to Ports A and B. Upon detecting the character H, the first digit has to be saved and displayed only when the second digit arrives. The desired program may be obtained by modifying the program in Figure 9.11 as follows:

```

/* Define register addresses */
#define RBUF (volatile char *) 0xFFFFFE0
#define SSTAT (volatile char *) 0xFFFFFE2
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define PBOUT (char *) 0xFFFFFFF4
#define PBDIR (char *) 0xFFFFFFF5

void main()
{
    /* Define the conversion table */
    char seg7[10] = {0x7E, 0x30, 0x6C, 0x79, 0x33, 0x5B, 0x5F, 0x30, 0x3F, 0x3B};
    char j, temp;

    /* Initialize the parallel ports */
    *PADIR = 0xFF; /* Configure Port A as output */
    *PBDIR = 0xFF; /* Configure Port B as output */

    /* Transfer the characters */
    while (1) { /* Infinite loop */
        while ((*SSTAT & 0x1) == 0); /* Wait for a new character */
        if (*RBUF == 'H') {
            while ((*SSTAT & 0x1) == 0); /* Wait for the first digit */
            j = *RBUF & 0xF; /* Extract the BCD value */
            temp = seg7[j]; /* Prepare 7-segment code for Port A */
            while ((*SSTAT & 0x1) == 0); /* Wait for the second digit */
            j = *RBUF & 0xF; /* Extract the BCD value */
            *PBOUT = seg7[j]; /* Send the 7-segment code to Port B */
            *PAOUT = temp; /* Send the 7-segment code to Port A */
        }
    }
}

```

9.6. The arrangement explained in the solutions for Problems 9.1 and 9.2 can be used, having the 7-segment displays connected to Ports A and B. Upon detecting the character H, the first digit has to be saved and displayed only when the second digit arrives. The desired program may be obtained by modifying the program in Figure 9.10 as follows:

|                               |          |                                                            |                                       |
|-------------------------------|----------|------------------------------------------------------------|---------------------------------------|
| RBUF                          | EQU      | \$FFFFFFE0                                                 | Receive buffer.                       |
| SSTAT                         | EQU      | \$FFFFFFE2                                                 | Status register for serial interface. |
| PAOUT                         | EQU      | \$FFFFFFF1                                                 | Port A output data.                   |
| PADIR                         | EQU      | \$FFFFFFF2                                                 | Port A direction register.            |
| PBOUT                         | EQU      | \$FFFFFFF4                                                 | Port B output data.                   |
| PBDIR                         | EQU      | \$FFFFFFF5                                                 | Port B direction register.            |
| * Define the conversion table |          |                                                            |                                       |
|                               | ORIGIN   | \$200                                                      |                                       |
| SEG7                          | DataByte | \$7E, \$30, \$6C, \$79, \$33, \$5B, \$5F, \$30, \$3F, \$3B |                                       |
| * Initialization              |          |                                                            |                                       |
|                               | ORIGIN   | \$1000                                                     |                                       |
|                               | MoveByte | #FF,PADIR                                                  | Configure Port A as output.           |
|                               | MoveByte | #FF,PBDIR                                                  | Configure Port B as output.           |
| * Transfer the characters     |          |                                                            |                                       |
| LOOP                          | Testbit  | #0,SSTAT                                                   | Check if new character is ready.      |
|                               | Branch=0 | LOOP                                                       |                                       |
|                               | MoveByte | RBUF,R0                                                    | Read the character.                   |
|                               | Compare  | #\$48,R0                                                   | Check if H.                           |
|                               | Branch≠0 | LOOP                                                       |                                       |
| LOOP2                         | Testbit  | #0,SSTAT                                                   | Check if first digit is ready.        |
|                               | Branch=0 | LOOP2                                                      |                                       |
|                               | MoveByte | RBUF,R0                                                    | Read the first digit.                 |
|                               | And      | #\$F,R0                                                    | Extract the BCD value.                |
| LOOP3                         | Testbit  | #0,SSTAT                                                   | Check if second digit is ready.       |
|                               | Branch=0 | LOOP3                                                      |                                       |
|                               | MoveByte | RBUF,R1                                                    | Read the second digit.                |
|                               | And      | #\$F,R1                                                    | Extract the BCD value.                |
|                               | MoveByte | SEG7(R1),PBOUT                                             | Send the 7-segment code to Port B.    |
|                               | MoveByte | SEG7(R0),PAOUT                                             | Send the 7-segment code to Port A.    |
|                               | Branch   | LOOP                                                       |                                       |



- 9.7. The arrangement explained in the solution for Problem 9.1 can be used, having the 7-segment displays connected to Ports A and B. Upon detecting the character H, the first digit has to be stored and displayed only when the second digit arrives. Interrupts are used to detect the arrival of both H and the subsequent pair of digits. Therefore, the interrupt service routine has to keep track of the received characters. Variable  $k$  is set to 2 when an H is detected, and it is decremented as the subsequent digits arrive. The desired program may be obtained by modifying the program in Figure 9.16 as follows:

```

/* Define register addresses */
#define RBUF (volatile char *) 0xFFFFFEE0
#define SCNT (char *) 0xFFFFFEE3
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define PBOUT (char *) 0xFFFFFFF4
#define PBDIR (char *) 0xFFFFFFF5
#define int_addr (int *) (0x24)

void intserv();

void main()
{
    /* Define the conversion table */
    char seg7[10] = {0x7E, 0x30, 0x6C, 0x79, 0x33, 0x5B, 0x5F, 0x30, 0x3F, 0x3B};
    char j;
    char digits[2];          /* Buffer for received BCD digits */
    int k = 0;               /* Set up to detect the first H */
    /* Initialize the parallel ports */
    *PADIR = 0xFF;           /* Configure Port A as output */
    *PBDIR = 0xFF;           /* Configure Port B as output */

    /* Initialize the interrupt mechanism */
    int_addr = &intserv;     /* Set interrupt vector */
    __asm__("Move #0x40,%PSR"); /* Processor responds to IRQ interrupts */
    *SCNT = 0x10;            /* Enable receiver interrupts */

    /* Transfer the characters */
    while (1);               /* Infinite loop */
}

```

```

/* Interrupt service routine */
void intserv()
{
    *SCONT = 0;           /* Disable interrupts */
    if (k > 0) {
        j = *RBUF & 0xF;  /* Extract the BCD value */
        k = k - 1;
        digits[k] = seg7[j]; /* Save 7-segment code for new digit */
        if (k == 0) {
            *PAOUT = digits[1]; /* Send first digit to Port A */
            *PBOUT = digits[0]; /* Send second digit to Port B */
        }
        else if (*RBUF == 'H') k = 2;
    }
    *SCONT = 0x10;        /* Enable receiver interrupts */
    __asm__("ReturnI");    /* Return from interrupt */
}

```

- 9.8. The arrangement explained in the solutions for Problems 9.1 and 9.2 can be used, having the 7-segment displays connected to Ports A and B. Upon detecting the character H, the first digit has to be stored and displayed only when the second digit arrives. Interrupts are used to detect the arrival of both H and the subsequent pair of digits. Therefore, the interrupt service routine has to keep track of the received characters. Variable *K* is set to 2 when an H is detected, and it is decremented as the subsequent digits arrive. The desired program may be obtained by modifying the program in Figure 9.14 as follows:

|       |     |            |                                   |
|-------|-----|------------|-----------------------------------|
| RBUF  | EQU | \$FFFFFFE0 | Receive buffer.                   |
| SCONT | EQU | \$FFFFFFE3 | Control reg for serial interface. |
| PAOUT | EQU | \$FFFFFFF1 | Port A output data.               |
| PADIR | EQU | \$FFFFFFF2 | Port A direction register.        |
| PBOUT | EQU | \$FFFFFFF4 | Port B output data.               |
| PBDIR | EQU | \$FFFFFFF5 | Port B direction register.        |

\* Define the conversion table and buffer for first digit

|      |             |                                                            |                           |
|------|-------------|------------------------------------------------------------|---------------------------|
|      | ORIGIN      | \$200                                                      |                           |
| SEG7 | DataByte    | \$7E, \$30, \$6C, \$79, \$33, \$5B, \$5F, \$30, \$3F, \$3B |                           |
| DIG  | ReserveByte | 1                                                          | Buffer for first digit.   |
| K    | Data        | 0                                                          | Set up to detect first H. |

\* Initialization

|  |          |               |                             |
|--|----------|---------------|-----------------------------|
|  | ORIGIN   | \$1000        |                             |
|  | MoveByte | #\$FF,PADIR   | Configure Port A as output. |
|  | MoveByte | #\$FF,PBDIR   | Configure Port B as output. |
|  | Move     | #INTSERV,\$24 | Set the interrupt vector.   |
|  | Move     | #\$40,PSR     | Processor responds to IRQ.  |
|  | MoveByte | #\$10,SCONT   | Enable receiver interrupts. |

\* Transfer loop

|      |        |      |                     |
|------|--------|------|---------------------|
| LOOP | Branch | LOOP | Infinite wait loop. |
|------|--------|------|---------------------|

\* Interrupt service routine

|         |          |                |                                |
|---------|----------|----------------|--------------------------------|
| INTSERV | MoveByte | #0, SCONT      | Disable interrupts.            |
|         | MoveByte | RBUF,R0        | Read the character.            |
|         | Move     | K,R1           | See if a new digit             |
|         | Branch>0 | NEWDIG         | is expected.                   |
|         | Compare  | #\$48,R0       | Check if H.                    |
|         | Branch≠0 | DONE           |                                |
|         | Move     | #2,K           | Detected an H.                 |
|         | Branch   | DONE           |                                |
| NEWDIG  | And      | #\$F,R0        | Extract the BCD value.         |
|         | Subtract | #1,R1          | Decrement K.                   |
|         | Move     | R1,K           |                                |
|         | Branch=0 | DISP           | Second digit received.         |
|         | MoveByte | SEG7(R0),DIG   | Save the first digit.          |
|         | Branch   | DONE           |                                |
| DISP    | MoveByte | DIG,PAOUT      | Send 7-segment code to Port A. |
|         | MoveByte | SEG7(R0),PBOUT | Send 7-segment code to Port B. |
| DONE    | MoveByte | #\$10,SCONT    | Enable receiver interrupts.    |
|         | ReturnI  |                | Return from interrupt.         |

- 9.9. Connect the parallel ports A and B to the four BCD to 7-segment decoders. Choose that  $PA_{7-4}$ ,  $PA_{3-0}$ ,  $PB_{7-4}$  and  $PB_{3-0}$  display the first, second, third and fourth received digits, respectively. Assume that all four digits arrive immediately after the character H has been received. The task can be achieved by modifying the program in Figure 9.11 as follows:

```

/* Define register addresses */
#define RBUF (volatile char *) 0xFFFFFE0
#define SSTAT (volatile char *) 0xFFFFFE2
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define PBOUT (char *) 0xFFFFFFF4
#define PBDIR (char *) 0xFFFFFFF5

void main()
{
    char temp;
    char digits[4];           /* Buffer for received digits */
    int i;
    /* Initialize the parallel ports */
    *PADIR = 0xFF;           /* Configure Port A as output */
    *PBDIR = 0xFF;           /* Configure Port B as output */

    /* Transfer the characters */
    while (1) {               /* Infinite loop */
        while ((*SSTAT & 0x1) == 0); /* Wait for a new character */
        if (*RBUF == 'H') {
            for (i = 3; i >= 0; i--) {
                while ((*SSTAT & 0x1) == 0); /* Wait for the next digit */
                digits[i] = *RBUF;           /* Save the new digit (ASCII) */
            }
            temp = digits[3] << 4;           /* Shift left first digit by 4 bits, */
            *PAOUT = temp | (digits[2] & 0xF); /* append second and send to A */
            temp = digits[1] << 4;           /* Shift left third digit by 4 bits, */
            *PBOUT = temp | (digits[0] & 0xF); /* append fourth and send to B */
        }
    }
}

```

9.10. Connect the parallel ports A and B to the four BCD to 7-segment decoders. Choose that  $PA_{7-4}$ ,  $PA_{3-0}$ ,  $PB_{7-4}$  and  $PB_{3-0}$  display the first, second, third and fourth received digits, respectively. Assume that all four digits arrive immediately after the character H has been received. Then, the desired program may be obtained by modifying the program in Figure 9.10 as follows:

|                           |                 |             |                                       |
|---------------------------|-----------------|-------------|---------------------------------------|
| RBUF                      | EQU             | \$FFFFFFE0  | Receive buffer.                       |
| SSTAT                     | EQU             | \$FFFFFFE2  | Status register for serial interface. |
| PAOUT                     | EQU             | \$FFFFFFF1  | Port A output data.                   |
| PADIR                     | EQU             | \$FFFFFFF2  | Port A direction register.            |
| PBOUT                     | EQU             | \$FFFFFFF4  | Port B output data.                   |
| PBDIR                     | EQU             | \$FFFFFFF5  | Port B direction register.            |
| * Initialization          |                 |             |                                       |
|                           | ORIGIN          | \$1000      |                                       |
|                           | MoveByte        | #\$FF,PADIR | Configure Port A as output.           |
|                           | MoveByte        | #\$FF,PBDIR | Configure Port B as output.           |
| * Transfer the characters |                 |             |                                       |
| LOOP                      | Testbit         | #0,SSTAT    | Check if new character is ready.      |
|                           | Branch=0        | LOOP        |                                       |
|                           | MoveByte        | RBUF,R0     | Read the character.                   |
|                           | Compare         | #\$48,R0    | Check if H.                           |
|                           | Branch $\neq$ 0 | LOOP        |                                       |
| LOOP2                     | Testbit         | #0,SSTAT    | Check if first digit is ready.        |
|                           | Branch=0        | LOOP2       |                                       |
|                           | MoveByte        | RBUF,R0     | Read the first digit.                 |
|                           | LShiftL         | #4,R0       | Shift left 4 bit positions.           |
| LOOP3                     | Testbit         | #0,SSTAT    | Check if second digit is ready.       |
|                           | Branch=0        | LOOP3       |                                       |
|                           | MoveByte        | RBUF,R1     | Read the second digit.                |
|                           | And             | #\$F,R1     | Extract the BCD value.                |
|                           | Or              | R1,R0       | Concatenate digits for Port A.        |
| LOOP4                     | Testbit         | #0,SSTAT    | Check if third digit is ready.        |
|                           | Branch=0        | LOOP4       |                                       |
|                           | MoveByte        | RBUF,R1     | Read the third digit.                 |
|                           | LShiftL         | #4,R1       | Shift left 4 bit positions.           |
| LOOP5                     | Testbit         | #0,SSTAT    | Check if fourth digit is ready.       |
|                           | Branch=0        | LOOP5       |                                       |
|                           | MoveByte        | RBUF,R2     | Read the fourth digit.                |
|                           | And             | #\$F,R2     | Extract the BCD value.                |
|                           | Or              | R2,R1       | Concatenate digits for Port B.        |
|                           | MoveByte        | R0,PAOUT    | Send digits to Port A.                |
|                           | MoveByte        | R1,PBOUT    | Send digits to Port B.                |
|                           | Branch          | LOOP        |                                       |

- 9.11. Connect the parallel ports A and B to the four BCD to 7-segment decoders. Choose that  $PA_{7-4}$ ,  $PA_{3-0}$ ,  $PB_{7-4}$  and  $PB_{3-0}$  display the first, second, third and fourth received digits, respectively. Upon detecting the character H, the subsequent four digits have to be saved and displayed only when the fourth digit arrives. Interrupts are used to detect the arrival of both H and the four digits. Therefore, the interrupt service routine has to keep track of the received characters. Variable  $k$  is set to 4 when an H is detected, and it is decremented as the subsequent digits arrive. The desired program may be obtained by modifying the program in Figure 9.16 as follows:

```

/* Define register addresses */
#define RBUF (volatile char *) 0xFFFFFE0
#define SCNT (char *) 0xFFFFFE3
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define PBOUT (char *) 0xFFFFFFF4
#define PBDIR (char *) 0xFFFFFFF5
#define int_addr (int *) (0x24)

void intserv();

void main()
{
    /* Define the conversion table */
    char seg7[10] = {0x7E, 0x30, 0x6C, 0x79, 0x33, 0x5B, 0x5F, 0x30, 0x3F, 0x3B};
    char temp;
    char digits[4];
    int k = 0;
    /* Initialize the parallel ports */
    *PADIR = 0xFF;
    *PBDIR = 0xFF;
    /* Configure Port A as output */
    /* Configure Port B as output */

    /* Initialize the interrupt mechanism */
    int_addr = &intserv;
    __asm__("Move #0x40,%PSR");
    *SCNT = 0x10;
    /* Set interrupt vector */
    /* Processor responds to IRQ interrupts */
    /* Enable receiver interrupts */

    /* Transfer the characters */
    while (1);
}

```

```

/* Interrupt service routine */
void intserv()
{
    *SCONT = 0; /* Disable interrupts */
    if (k > 0) {
        k = k - 1;
        digits[k] = *RBUF; /* Save the new digit (ASCII) */
        if (k == 0) {
            temp = digits[3] << 4; /* Shift left first digit by 4 bits, */
            *PAOUT = temp | (digits[2] & 0xF); /* append second and send to A */
            temp = digits[1] << 4; /* Shift left third digit by 4 bits */
            *PBOUT = temp | (digits[0] & 0xF); /* append fourth and send to B */
        }
        else if (*RBUF == 'H') k = 4;
    }
    *SCONT = 0x10; /* Enable receiver interrupts */
    __asm__("ReturnI"); /* Return from interrupt */
}

```

- 9.12. Connect the parallel ports A and B to the four BCD to 7-segment decoders. Choose that  $PA_{7-4}$ ,  $PA_{3-0}$ ,  $PB_{7-4}$  and  $PB_{3-0}$  display the first, second, third and fourth received digits, respectively. Upon detecting the character H, the subsequent four digits have to be saved and displayed only when the fourth digit arrives. Interrupts are used to detect the arrival of both H and the four digits. Therefore, the interrupt service routine has to keep track of the received characters. Variable  $K$  is set to 4 when an H is detected, and it is decremented as the subsequent digits arrive. The desired program may be obtained by modifying the program in Figure 9.14 as follows:

|                             |             |               |                                   |
|-----------------------------|-------------|---------------|-----------------------------------|
| RBUF                        | EQU         | \$FFFFFFE0    | Receive buffer.                   |
| SCONT                       | EQU         | \$FFFFFFE3    | Control reg for serial interface. |
| PAOUT                       | EQU         | \$FFFFFFF1    | Port A output data.               |
| PADIR                       | EQU         | \$FFFFFFF2    | Port A direction register.        |
| PBOUT                       | EQU         | \$FFFFFFF4    | Port B output data.               |
| PBDIR                       | EQU         | \$FFFFFFF5    | Port B direction register.        |
|                             | ORIGIN      | \$200         |                                   |
| DIG                         | ReserveByte | 4             | Buffer for received digits.       |
| K                           | Data        | 0             | Set up to detect first H.         |
| * Initialization            |             |               |                                   |
|                             | ORIGIN      | \$1000        |                                   |
|                             | MoveByte    | #\$FF,PADIR   | Configure Port A as output.       |
|                             | MoveByte    | #\$FF,PBDIR   | Configure Port B as output.       |
|                             | Move        | #INTSERV,\$24 | Set the interrupt vector.         |
|                             | Move        | #\$40,PSR     | Processor responds to IRQ.        |
|                             | MoveByte    | #\$10,SCONT   | Enable receiver interrupts.       |
| * Transfer loop             |             |               |                                   |
| LOOP                        | Branch      | LOOP          | Infinite wait loop.               |
| * Interrupt service routine |             |               |                                   |
| INTSERV                     | MoveByte    | #0,SCONT      | Disable interrupts.               |
|                             | MoveByte    | RBUF,R0       | Read the character.               |
|                             | Move        | K,R1          | See if a new digit                |
|                             | Branch>0    | NEWDIG        | is expected.                      |
|                             | Compare     | #\$48,R0      | Check if H.                       |
|                             | Branch≠0    | DONE          |                                   |
|                             | Move        | #4,K          | Detected an H.                    |
|                             | Branch      | DONE          |                                   |
| NEWDIG                      | And         | #\$F,R0       | Extract the BCD value.            |
|                             | Subtract    | #1,R1         | Decrement K.                      |
|                             | MoveByte    | R0,DIG(R1)    | Save the digit.                   |
|                             | Move        | R1,K          |                                   |
|                             | Branch>0    | DONE          | Expect more digits.               |
|                             | Move        | #DIG,R0       | Pointer to buffer for digits.     |
| DISP                        | MoveByte    | (R0)+,R1      | Get fourth digit.                 |
|                             | MoveByte    | (R0)+,R2      | Get third digit and               |
|                             | LShiftL     | #4,R2         | shift it left.                    |
|                             | Or          | R1,R2         | Concatenate digits for Port B.    |
|                             | MoveByte    | R2,PBOUT      | Send digits to Port B.            |
|                             | MoveByte    | (R0)+,R1      | Get second digit.                 |
|                             | MoveByte    | (R0)+,R2      | Get first digit and               |
|                             | LShiftL     | #4,R2         | shift it left.                    |
|                             | Or          | R1,R2         | Concatenate digits for Port A.    |
|                             | MoveByte    | R2,PAOUT      | Send digits to Port A.            |
| DONE                        | MoveByte    | #\$10,SCONT   | Enable receiver interrupts.       |
|                             | ReturnI     |               | Return from interrupt.            |



- 9.13. Use a table to convert a received ASCII digit into a 7-segment code as explained in the solution for Problem 9.1. Connect the bits  $S_a$  to  $S_g$  of all four registers to bits  $PA_{6-0}$  of Port A. Use bits  $PB_3$  to  $PB_0$  of Port B as Load signals for the registers displaying the first, second, third and fourth received digits, respectively. Then, the required task can be achieved by modifying the program in Figure 9.11 as follows:

```

/* Define register addresses */
#define RBUF (volatile char *) 0xFFFFFEE0
#define SSTAT (volatile char *) 0xFFFFFEE2
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define PBOUT (char *) 0xFFFFFFF4
#define PBDIR (char *) 0xFFFFFFF5

void main()
{
    /* Define the conversion table */
    char seg7[10] = {0x7E, 0x30, 0x6C, 0x79, 0x33, 0x5B, 0x5F, 0x30, 0x3F, 0x3B};
    char j;
    char digits[4];
    int k = 0;
    int i;

    /* Initialize the parallel ports */
    *PADIR = 0xFF;
    *PBDIR = 0xFF;

    /* Transfer the characters */
    while (1) {
        while ((*SSTAT & 0x1) == 0);
        if (*RBUF == 'H') {
            for (i = 3; i >= 0; i--) {
                while ((*SSTAT & 0x1) == 0);
                j = *RBUF & 0xF;
                digits[i] = seg7[j];
            }
            for (i = 0; i <= 3; i++) {
                *PAOUT = digits[i];
                *PBOUT = 1 << i;
                *PBOUT = 0;
            }
        }
    }
}

```

- 9.14. Use a table to convert a received ASCII digit into a 7-segment code as explained in the solution for Problem 9.1. Connect the bits  $S_a$  to  $S_g$  of all four registers to bits  $PA_{6-0}$  of Port A. Use bits  $PB_3$  to  $PB_0$  of Port B as Load signals for the registers displaying the first, second, third and fourth received digits, respectively. Then, the required task can be achieved by modifying the program in Figure 9.10 as follows:

```

RBUF    EQU    $FFFFFFE0    Receive buffer.
SSTAT   EQU    $FFFFFFE2    Status register for serial interface.
PAOUT   EQU    $FFFFFFF1    Port A output data.
PADIR   EQU    $FFFFFFF2    Port A direction register.
PBOUT   EQU    $FFFFFFF4    Port B output data.
PBDIR   EQU    $FFFFFFF5    Port B direction register.

* Define the conversion table and buffer for received digits
        ORIGIN    $200
SEG7    DataByte  $7E, $30, $6C, $79, $33, $5B, $5F, $30, $3F, $3B
DIG     ReserveByte 4                Buffer for received digits.

* Initialization
        ORIGIN    $1000
        MoveByte  #$FF, PADIR        Configure Port A as output.
        MoveByte  #$FF, PBDIR        Configure Port B as output.

* Transfer the characters
LOOP    Testbit    #0, SSTAT          Check if new character is ready.
        Branch=0   LOOP
        MoveByte   RBUF, R0           Read the character.
        Compare    #48, R0            Check if H.
        Branch≠0   LOOP
        Move       #3, R1             Set up a counter.
LOOP2   Testbit    #0, SSTAT          Check if next digit is available.
        Branch=0   LOOP2
        MoveByte   RBUF, R0           Read the digit.
        And        #4, R0             Extract the BCD value.
        MoveByte   SEG7(R0), DIG(R1)  Save 7-seg code for the digit.
        Subtract   #1, R1             Check if more digits
        Branch>=0   LOOP2             are expected.
        Move       #DIG, R0           Pointer to buffer for digits.
        Move       #8, R1             Set up Load signal for  $d_3$ .
DISP    MoveByte   (R0)+, PAOUT        Send 7-segment code to Port A.
        MoveByte   R1, PBOUT          Load the digit into its register.
        MoveByte   #0, PBOUT          Clear the Load signal.
        LShiftR    #1, R1            Set Load for the next digit.
        Branch>0   DISP              There are more digits to send.
        Branch     LOOP

```

- 9.15. Use a table to convert a received ASCII digit into a 7-segment code as explained in the solutions for Problems 9.1. and 9.2. Connect the bits  $S_a$  to  $S_g$  of all four registers to bits  $PA_{6-0}$  of Port A. Use bits  $PB_3$  to  $PB_0$  of Port B as Load signals for the registers displaying the first, second, third and fourth received digits, respectively. Upon detecting the character H, the subsequent four digits have to be saved and displayed only when the fourth digit arrives. Interrupts are used to detect the arrival of both H and the four digits. Therefore, the interrupt service routine has to keep track of the received characters. Variable  $k$  is set to 4 when an H is detected, and it is decremented as the subsequent digits arrive. The desired program may be obtained by modifying the program in Figure 9.16 as follows:

```

/* Define register addresses */
#define RBUF (volatile char *) 0xFFFFFE0
#define SCNT (char *) 0xFFFFFE3
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define PBOUT (char *) 0xFFFFFFF4
#define PBDIR (char *) 0xFFFFFFF5
#define int_addr (int *) (0x24)

void intserv();

void main()
{
    /* Define the conversion table */
    char seg7[10] = {0x7E, 0x30, 0x6C, 0x79, 0x33, 0x5B, 0x5F, 0x30, 0x3F, 0x3B};
    char j;
    char digits[4];          /* Buffer for received BCD digits */
    int k = 0;               /* Set up to detect the first H */
    int i;

    /* Initialize the parallel ports */
    *PADIR = 0xFF;           /* Configure Port A as output */
    *PBDIR = 0xFF;           /* Configure Port B as output */

    /* Initialize the interrupt mechanism */
    int_addr = &intserv;     /* Set interrupt vector */
    __asm__ ("Move #0x40,%PSR"); /* Processor responds to IRQ interrupts */
    *SCNT = 0x10;            /* Enable receiver interrupts */

    /* Transfer the characters */
    while (1);               /* Infinite loop */
}

```

```

/* Interrupt service routine */
void intserv()
{
    *SCONT = 0;                /* Disable interrupts */
    if (k > 0) {
        j = *RBUF & 0xF;      /* Extract the BCD value */
        k = k - 1;
        digits[k] = seg7[j];  /* Save 7-segment code for new digit */
        if (k == 0) {
            for (i = 0; i <= 3; i++) {
                *PAOUT = digits[i]; /* Send a digit to Port A */
                *PBOUT = 1 << i;    /* Load the digit into its register */
                *PBOUT = 0;          /* Clear the Load signal */
            }
        }
        else if (*RBUF == 'H') k = 4;
    }
    *SCONT = 0x10;             /* Enable receiver interrupts */
    _asm_ ("ReturnI");         /* Return from interrupt */
}

```

- 9.16. Use a table to convert a received ASCII digit into a 7-segment code as explained in the solutions for Problems 9.1. and 9.2. Connect the bits  $S_a$  to  $S_g$  of all four registers to bits  $PA_{6-0}$  of Port A. Use bits  $PB_3$  to  $PB_0$  of Port B as Load signals for the registers displaying the first, second, third and fourth received digits, respectively. Upon detecting the character H, the subsequent four digits have to be saved and displayed only when the fourth digit arrives. Interrupts are used to detect the arrival of both H and the four digits. Therefore, the interrupt service routine has to keep track of the received characters. Variable  $K$  is set to 4 when an H is detected, and it is decremented as the subsequent digits arrive. The desired program may be obtained by modifying the program in Figure 9.14 as follows:

|       |     |            |                                   |
|-------|-----|------------|-----------------------------------|
| RBUF  | EQU | \$FFFFFFE0 | Receive buffer.                   |
| SCONT | EQU | \$FFFFFFE3 | Control reg for serial interface. |
| PAOUT | EQU | \$FFFFFFF1 | Port A output data.               |
| PADIR | EQU | \$FFFFFFF2 | Port A direction register.        |
| PBOUT | EQU | \$FFFFFFF4 | Port B output data.               |
| PBDIR | EQU | \$FFFFFFF5 | Port B direction register.        |

\* Define the conversion table and buffer for received digits

|      |             |                                                            |                             |
|------|-------------|------------------------------------------------------------|-----------------------------|
|      | ORIGIN      | \$200                                                      |                             |
| SEG7 | DataByte    | \$7E, \$30, \$6C, \$79, \$33, \$5B, \$5F, \$30, \$3F, \$3B |                             |
| DIG  | ReserveByte | 4                                                          | Buffer for received digits. |
| K    | Data        | 0                                                          | Set up to detect first H.   |

\* Initialization

|  |          |               |                             |
|--|----------|---------------|-----------------------------|
|  | ORIGIN   | \$1000        |                             |
|  | MoveByte | #\$FF,PADIR   | Configure Port A as output. |
|  | MoveByte | #\$FF,PBDIR   | Configure Port B as output. |
|  | Move     | #INTSERV,\$24 | Set the interrupt vector.   |
|  | Move     | #\$40,PSR     | Processor responds to IRQ.  |
|  | MoveByte | #\$10,SCONT   | Enable receiver interrupts. |

\* Transfer loop

|      |        |      |                     |
|------|--------|------|---------------------|
| LOOP | Branch | LOOP | Infinite wait loop. |
|------|--------|------|---------------------|

\* Interrupt service routine

|         |          |                  |                                   |
|---------|----------|------------------|-----------------------------------|
| INTSERV | MoveByte | #0, SCONT        | Disable interrupts.               |
|         | MoveByte | RBUF,R0          | Read the character.               |
|         | Move     | K,R1             | See if a new digit                |
|         | Branch>0 | NEWDIG           | is expected.                      |
|         | Compare  | #\$48,R0         | Check if H.                       |
|         | Branch≠0 | DONE             |                                   |
|         | Move     | #4,K             | Detected an H.                    |
|         | Branch   | DONE             |                                   |
| NEWDIG  | And      | #\$F,R0          | Extract the BCD value.            |
|         | Subtract | #1,R1            | Decrement K.                      |
|         | MoveByte | SEG7(R0),DIG(R1) | Save 7-seg code for the digit.    |
|         | Move     | R1,K             |                                   |
|         | Branch>0 | DONE             | Expect more digits.               |
|         | Move     | #DIG,R0          | Pointer to buffer for digits.     |
|         | Move     | #8,R1            | Set up Load signal for $d_3$ .    |
| DISP    | MoveByte | (R0)+,PAOUT      | Send 7-segment code to Port A.    |
|         | MoveByte | R1,PBOUT         | Load the digit into its register. |
|         | MoveByte | #0,PBOUT         | Clear the Load signal.            |
|         | LShiftR  | #1,R1            | Set Load for the next digit.      |
|         | Branch>0 | DISP             | There are more digits to send.    |
| DONE    | MoveByte | #\$10,SCONT      | Enable receiver interrupts.       |
|         | ReturnI  |                  | Return from interrupt.            |

9.17. Programs in Figures 9.17 and 9.18 would not work properly if the circular buffer was filled with 80 characters. After the head pointer wraps around, it would trail the tail pointer and would catch up with it if the buffer is full. At this point it would be impossible to use the simple comparison of the two pointers to determine whether the buffer is empty or full. The simplest modification is to increase the buffer size to 81 characters.

9.18. Using a counter variable,  $M$ , the program in Figure 9.17 can be modified as follows:

```

/* Define register addresses */
#define RBUF (volatile char *) 0xFFFFFEE0
#define SSTAT (volatile char *) 0xFFFFFEE2
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define PSTAT (volatile char *) 0xFFFFFFF6
#define BSIZE 80

void main()
{
    unsigned char mbuffer[BSIZE];
    unsigned char fin, fout;
    unsigned char temp;
    int M = 0;

    /* Initialize Port A and circular buffer */
    *PADIR = 0xFF; /* Configure Port A as output */
    fin = 0;
    fout = 0;

    /* Transfer the characters */
    while (1) { /* Infinite loop */
        while ((*SSTAT & 0x1) == 0) { /* Wait for a new character */
            if (M > 0) { /* If circular buffer is not empty */
                if (*PSTAT & 0x2) { /* and output device is ready */
                    *PAOUT = mbuffer[fout]; /* send a character to Port A */
                    M = M - 1; /* Decrement the queue counter */
                    if (fout < BSIZE-1) /* Update the output index */
                        fout++;
                    else
                        fout = 0;
                }
            }
        }
        mbuffer[fin] = *RBUF; /* Read a character from receive buffer */
        M = M + 1; /* Increment the queue counter */
        if (fin < BSIZE-1) /* Update the input index */
            fin++;
        else
            fin = 0;
    }
}

```

9.19. Using a counter variable, *M*, the program in Figure 9.18 can be modified as follows:

|                           |                 |               |                                    |
|---------------------------|-----------------|---------------|------------------------------------|
| RBUF                      | EQU             | \$FFFFFFE0    | Receive buffer.                    |
| SSTAT                     | EQU             | \$FFFFFFE2    | Status reg for serial interface.   |
| PAOUT                     | EQU             | \$FFFFFFF1    | Port A output data.                |
| PADIR                     | EQU             | \$FFFFFFF2    | Port A direction register.         |
| PSTAT                     | EQU             | \$FFFFFFF6    | Status reg for parallel interface. |
| MBUF                      | ReserveByte     | 80            | Define the circular buffer.        |
| * Initialization          |                 |               |                                    |
|                           | ORIGIN          | \$1000        |                                    |
|                           | MoveByte        | #\$FF,PADIR   | Configure Port A as output.        |
|                           | Move            | #MBUF,R0      | R0 points to the buffer.           |
|                           | Move            | #0,R1         | Initialize head pointer.           |
|                           | Move            | #0,R2         | Initialize tail pointer.           |
|                           | Move            | #0,R3         | Initialize queue counter.          |
| * Transfer the characters |                 |               |                                    |
| LOOP                      | Testbit         | #0,SSTAT      | Check if new character is ready.   |
|                           | Branch $\neq$ 0 | READ          |                                    |
|                           | Compare         | #0,R3         | Check if queue is empty.           |
|                           | Branch=0        | LOOP          | Queue is empty.                    |
|                           | Testbit         | #1,PSTAT      | Check if Port A is ready.          |
|                           | Branch=0        | LOOP          |                                    |
|                           | MoveByte        | (R0,R2),PAOUT | Send a character to Port A.        |
|                           | Subtract        | #1,R3         | Decrement the queue counter.       |
|                           | Add             | #1,R2         | Increment the tail pointer.        |
|                           | Compare         | #80,R2        | Is the pointer past queue limit?   |
|                           | Branch<0        | LOOP          |                                    |
|                           | Move            | #0,R2         | Wrap around.                       |
|                           | Branch          | LOOP          |                                    |
| READ                      | MoveByte        | RBUF,(R0,R1)  | Place new character into queue.    |
|                           | Add             | #1,R3         | Increment the queue counter.       |
|                           | Add             | #1,R1         | Increment the head pointer.        |
|                           | Compare         | #80,R1        | Is the pointer past queue limit?   |
|                           | Branch<0        | LOOP          |                                    |
|                           | Move            | #0,R1         | Wrap around.                       |
|                           | Branch          | LOOP          |                                    |

- 9.20. Connect the two 7-segment displays to Port A. Use the 3 bits of Port B to connect to the switches and LED as shown in Figure 9.19. It is necessary to modify the conversion and display portions of programs in Figures 9.20 and 9.21.

The end of the program in Figure 9.20 should be:

```

/* Compute the total count */
total_count = (0xFFFFFFFF - counter_value);

/* Convert count to time */ ;
actual_time = total_count / 1000000;      /* Time in hundredths of seconds */
tenths = actual_time / 10;
hundredths = actual_time - tenths * 10;

*PAOUT = ((tenths << 4) | hundredths); /* Display the elapsed time */
}

```

The end of the program in Figure 9.20 should be:

```

* Convert the count to actual time in hundredths of seconds,
* and then to BCD. Put the BCD digits in R4.
Move      #1000000,R1    Determine the count in
Divide     R1,R2          hundredths of seconds.
Move      #10,R1         Divide by 10 to find the digit that
Divide     R1,R2          denotes 1/10th of a second.
LShiftL    #4,R3         The BCD digits
Or         R2,R3          are placed in R3.

MoveByte   R3,PAOUT      Send digits to Port A.
Branch     START         Ready for next test.

```



# Chapter 10 – Computer Peripherals

- 10.1. *Revised problem statement:* The total time required to illuminate each pixel on the display screen of a computer monitor is 5ns. The beam is then turned off and moved to the next point to be illuminated. On average, moving the beam from one spot to the next takes 12 ns. What is the maximum possible resolution of this display if it is to be refreshed 70 times per second.

For  $N$  pixels we get

$$\text{Scan time} = (5 + 12)N = 10^9/70 = 14.3\text{ns}$$

Hence,  $N = 840000$  pixels

A commercial standard that would not exceed this resolution is  $1024 \times 768$ .

- 10.2. Each symbol can have one of eight possible values, which means it represents three bits. Therefore:

$$\text{Bit rate} = 3 \times \text{baud rate} = 3 \times 9600 = 28,800 \text{ bits/s}$$

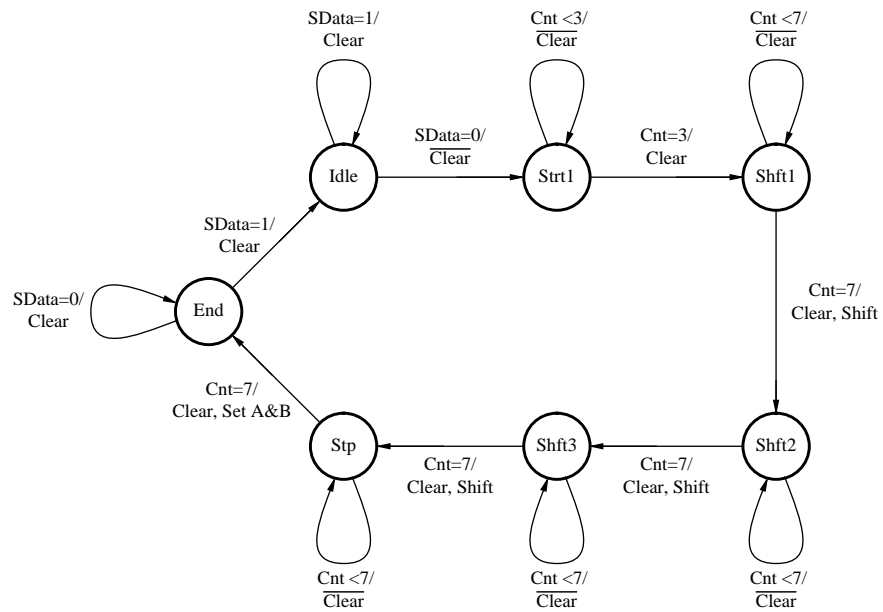
- 10.3. In preparing this design we have assumed the following:

- The counter has a synchronous Clear signal. That is, the counter is cleared to 0 on the clock edge at the end of a clock period during which Clear = 1.
- The shift register has a synchronous control signal called Shift. The data value at its serial input is shifted into the register on the clock edge at the end of a clock period during which Shift = 1.

We use a D flip-flop as a synchronizer for the input data. Its output, SData, follows the input data, but is synchronized with the local clock. It is connected to the serial input of the shift register. Both the shift register and the counter are driven by the local clock.

We will now describe the control logic that generates the Clear and Shift signals. Starting from an idle state in which SData = 1, Clear = 1, and Shift = 0, the sequence of events that the control logic needs to implement is as follows:

- (a) When SData = 0 change Clear to 0. The counter starts to count.
- (b) When count = 3 (the fourth clock cycle), set Clear = 1 for one clock cycle. The clock edge at the end of this cycle is the mid-point of the Start bit. The counter is cleared to 0 at this point, then it starts to count again.
- (c) When count reaches 7, set both Clear and Shift to 1 for one clock cycle. At the end of this clock cycle, the first data bit is loaded in the shift register and the counter is again cleared to 0. Repeat twice.
- (d) When count = 7, set A = SData and B =  $\overline{\text{SData}}$ .



(e) Wait until SData = 1 then return to step 1.

A state diagram for the control logic is given below. When not specified, outputs are equal to zero.

10.4 Each data byte requires 10 bits to transmit. Hence, the effective transmission rate is  $38,800/10 = 3,800$  bytes/s.

10.5 A: 1100 0001, P: 0101 0000, =: 0011 1101, 5: 1011 0101

10.6 (Correction: Bit  $b_7$  is the Data Set Ready signal, CC).

We will refer to the register given in the problem as STATUS. The program below deals with an incoming call.

|                                                                                                |          |            |                            |
|------------------------------------------------------------------------------------------------|----------|------------|----------------------------|
|                                                                                                | BitSet   | #1,STATUS  | Enable automatic answering |
| RING                                                                                           | BitTest  | #14,Status | Wait for ringing signal    |
|                                                                                                | Branch=0 | RING       |                            |
| * At this point, the program may alert the user (or the operating-system) of an in-coming call |          |            |                            |
| Ready                                                                                          | BitTest  | #7,Status  | Wait for Data Set Ready    |
|                                                                                                | Branch=0 | Ready      |                            |
|                                                                                                | BitSet   | #2,STATUS  | Enable send carrier        |
| SENDC                                                                                          | BitTest  | #13,STATUS | Wait for confirmation      |
|                                                                                                | Branch=0 | SENDC      |                            |
| RECV                                                                                           | BitTest  | #12,STATUS | Wait for receive carrier   |
|                                                                                                | Branch=0 | RECV       |                            |
| * Program is now ready to send and receive data                                                |          |            |                            |

# Chapter 11

## Processor Families

- 11.1. The main ideas of conditional execution of ARM instructions (see Sections 3.1.2 and B.1) and conditional execution of IA-64 instructions, called predication (see Section 11.7.2), are very similar.

The differences occur in the way that the conditions are set and stored in the processor, and in the way that they are referenced by the conditionally executed instructions.

In ARM processors, the state is stored in four conventional condition code flags N, Z, C, and V (see Section 3.1.1). These flags are optionally set by the results of instruction execution. The particular condition, which may be a function of more than one flag, is named in the condition field of each ARM instruction (see Figure B.1 and Table B.1).

In the IA-64 architecture, there are no conventional condition code flags. Instead, the result (true or false) of executing a Compare<condition> instruction is stored in one of 64 one-bit predicate registers, as described in Section 11.7.2. Each instruction can name one of these bits in its 6-bit predicate field; and the instruction is executed only if the bit is 1 (true).

- 11.2. Assume that Thumb arithmetic instructions have a 2-operand format, expressed in assembly language as

OP    *Rdst,Rsrc*

as discussed in Section 11.1.1

Also assume that a signed integer Divide instruction (DIV) is available in the Thumb instruction set with the assembly language format

DIV    *Rdst,Rsrc*

This instruction performs the operation  $[Rdst]/[Rsrc]$ . It stores the quotient in *Rdst* and stores the remainder in *Rsrc*.

Under these assumptions, a possible Thumb program would be:

```
LDR    R0,G
LDR    R1,H
ADD    R0,R1    Leaves  $g + h$  in R0.
LDR    R1,E
LDR    R2,F
MUL    R1,R2    Leaves  $e \times f$  in R1.
DIV    R1,R0    Leaves  $(e \times f)/(g + h)$  in R1.
LDR    R0,C
LDR    R2,D
DIV    R0,R2    Leaves  $c/d$  in R0.
ADD    R0,R1    Leaves denominator in R0.
LDR    R1,A
LDR    R2,B
ADD    R1,R2    Leaves  $a + b$  in R1.
DIV    R1,R0    Leaves result in R1.
STR    R1,W    Stores result in  $w$ .
```

This program requires 16 instructions as compared to 13 instruction words (some combined instructions) in the HP3000.

11.3. The following table shows some of the important areas for similarity/difference comparisons.

| MOTOROLA 680X0                                                                  | INTEL 80X86                                                        |
|---------------------------------------------------------------------------------|--------------------------------------------------------------------|
| 8 Data registers and 8 Address registers (including a processor stack register) | 8 General registers (including a processor stack register)         |
| CISC instruction set with flexible addressing modes                             | CISC instruction set with flexible addressing modes                |
| Large instruction set with multiple-register load/store instructions            | Large instruction set with multiple-register push/pop instructions |
| Memory-mapped I/O only                                                          | Separate I/O space as well as memory-mapped I/O                    |
| Flat address space                                                              | Segmented address space                                            |
| Big-endian addressing                                                           | Little-endian addressing                                           |

There is roughly comparable capability and performance between pairs from these two families; that is 68000 vs. 8086, 68020 vs. 80286, 68030 vs. 80386, and 68040 vs. 80486. The cache and pipelining aspects for the high end of each family are summarized in Sections 11.2.2 and 11.3.3.

- 11.4. An instruction cache is simpler to implement, because its entries do not have to be written back to the main memory. A data cache must have a provision for writing any changed entries back to the memory, before they are overwritten by new entries. From a performance standpoint, a single larger instruction cache would be advantageous only if the frequency of memory data accesses were very low. A unified cache has the potential performance advantage that the proportions of instructions and data vary automatically as a program is executed. However, if separate instruction and data caches are used, they can be accessed in parallel in a pipelined machine; and this is the major performance advantage.
- 11.5. Memory-mapped I/O requires no specialized support in terms of either instructions or bus signals. A separate I/O space allows simpler I/O interfaces and potentially faster operation. Processors such as those in the IA-32 family, that have a separate I/O space, can also use memory-mapped I/O.

- 11.6. **MOTOROLA** - The Autoincrement and Autodecrement modes facilitate stack implementation and accessing successive items in a list. Significant flexibility in accessing structured lists and arrays of addresses and data of different sizes is provided by the displacement, offset, and scale factor features, coupled with indirection.

**INTEL** - Relocatability in the physical address space is facilitated by the way in which base, index and displacement features are used in generating virtual addresses. As in the Motorola processors, these multiple-component address features enable flexible access to address lists and data structures.

In both families of processors, byte-addressability enables handling of character strings, and the Intel IA-32 String instructions (see Sections 3.21.3 and D.4.1) facilitate movement and processing of byte and doubleword data blocks. The Motorola MOVEM and MOVEP instructions perform similar operations.

- 11.7. *Flat address space* — Simplest configuration from the standpoint of a single user program and its compilation.

*One or more variable-length segments* — Efficient allocation of available memory space to variable-length user or operating system programs.

*Paged memory* — Facilitates automated memory management between the random-access main memory and a sector-organized disk secondary memory (see Chapters 5 and 10). Access privileges can be controlled on a page-by-page basis to ensure protection among users, and between users and the operating system when shared data are involved.

*Segmentation and paging* — Most flexible arrangement for managing multiple user and system address spaces, including protection mechanisms. The virtual address space can be significantly larger than the physical main memory space.

### 11.8. ARM program:

Assume that a signed integer Divide instruction is available in the ARM instruction set, and that it has the same format as the Multiply (MUL) instruction (see Figure B.4). The assembly language expression for the Divide (DIV) instruction is

DIV     $Rd, Rm, Rs$

and it performs the operation  $[Rm]/[Rs]$ , loading the quotient into  $Rm$  and the remainder into  $Rd$ .

|     |             |                           |
|-----|-------------|---------------------------|
| LDR | R0,C        |                           |
| LDR | R1,D        |                           |
| DIV | R2,R0,R1    | Leaves $c/d$ in R0.       |
| LDR | R1,G        |                           |
| LDR | R2,H        |                           |
| ADD | R1,R1,R2    | Leaves $g + h$ in R1.     |
| LDR | R2,F        |                           |
| DIV | R3,R2,R1    | Leaves $f/(g + h)$ in R2. |
| LDR | R3,E        |                           |
| MLA | R1,R2,R3,R0 | Leaves denominator in R1. |
| LDR | R0,A        |                           |
| LDR | R2,B        |                           |
| ADD | R0,R0,R2    | Leaves $a + b$ in R0.     |
| DIV | R2,R0,R1    | Leaves result in R0.      |
| STR | R0,W        | Stores result in $w$ .    |

This program requires 15 instructions as compared to 13 instruction words (some combined instructions) in the HP3000.



68000 program (assume 16-bit operands):

|       |       |                                      |
|-------|-------|--------------------------------------|
| MOVE  | G,D0  |                                      |
| ADD   | H,D0  | Leaves $g + h$ in D0.                |
| MOVE  | E,D1  |                                      |
| MULS  | F,D1  | Leaves $e \times f$ in D1.           |
| DIVS  | D0,D1 | Leaves $(e \times f)/(g + h)$ in D1. |
| MOVE  | C,D0  |                                      |
| EXT.L | D0    | See <i>Note</i> below.               |
| DIVS  | D,D0  | Leaves $c/d$ in D0.                  |
| ADD   | D1,D0 | Leaves denominator in D0.            |
| MOVE  | A,D1  |                                      |
| ADD   | B,D1  |                                      |
| EXT.L | D1    | See <i>Note</i> below.               |
| DIVS  | D0,D1 | Leaves result in D1.                 |
| MOVE  | D1,W  | Stores result in $w$ .               |

*Note:* The EXT.L instruction sign-extends the 16-bit dividend in the destination register to 32 bits, a requirement of the Divide instruction.

This program contains 14 instructions, as compared to 13 instruction words (some combined instructions) in the HP3000.

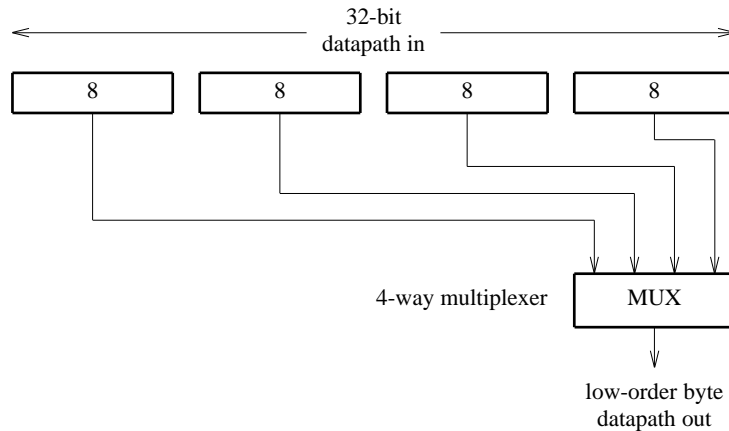
IA-32 program:

|      |         |                                       |
|------|---------|---------------------------------------|
| MOV  | EBX,G   |                                       |
| ADD  | EBX,H   | Leaves $g + h$ in EBX.                |
| MOV  | EAX,E   |                                       |
| IMUL | EAX,F   | Leaves $e \times f$ in EAX.           |
| CDQ  |         | See <i>Note</i> below.                |
| IDIV | EBX     |                                       |
| MOV  | EBX,EAX | Leaves $(e \times f)/(g + h)$ in EBX. |
| MOVE | EAX,C   |                                       |
| CDQ  |         | See <i>Note</i> below.                |
| IDIV | D       | Leaves $c/d$ in EAX.                  |
| ADD  | EBX,EAX | Leaves denominator in EBX.            |
| MOVE | EAX,A   |                                       |
| ADD  | EAX,B   | Leaves $a + b$ in EAX.                |
| CDQ  |         | See <i>Note</i> below.                |
| IDIV | EBX     | Leaves result in EAX.                 |
| MOV  | W,EAX   | Stores result in $w$ .                |

*Note:* The CDQ instruction sign-extends EAX into EDX (see Section 3.23.1), a requirement of the Divide instruction.

This program contains 16 instructions, as compared to 13 instruction words (some combined instructions) in the HP3000.

11.9. A 4-way multiplexer is required, as shown in the following figure.



- 11.10. There are no direct counterparts of the memory stack pointer registers SP and FP in the IA-64 architecture. The register remapping hardware in IA-64 processors allows the main program and any sequence of nested subroutines to all use logical register addresses R32 and upward for their own local variables, with the first part of that register space containing parameters passed from the calling routine. An example of this is shown in Figure 11.4.

If the 92 registers of the stacked physical register space are used up by register allocations for a sequence of nested subroutine calls, then some of those physical registers must be spilled into memory to create physical register space for any additional nested subroutines. The memory pointer register used by the processor for that memory area could be considered as a counterpart of SP; but it is not actually used as a TOS pointer by the current routine. In fact, it is not visible to user programs.

- 11.11. Consider the example of a main program calling a subroutine, as shown in Figure 11.4. The physical register addresses of registers used by the main program are the same as the logical register addresses used in the main program instructions. However, the logical register addresses above 31 used by instructions in the subroutine must have 8 added to them to generate the correct physical register addresses.

The value 8 is the first operand in the Alloc 8,4 instruction executed by the main program. When that instruction is executed, the value 8 is stored in a processor state register associated with the main program. After the subroutine is entered, all logical register addresses above 31 issued by its instructions must be added, in a small adder, to the value (8) in that register. The output of this adder is the physical register address to be used while in the subroutine.

The operand 7 in the Alloc 7,3 instruction executed by the subroutine is stored in a second processor state register associated with the subroutine. The output of that register is added in a second adder to the output of the first adder. After the subroutine calls a second subroutine, logical register addresses above 31 issued by the second subroutine are sent into the first adder. The output of the second adder (logical address + 8 + 7) is the physical register address used while in the second subroutine.

More register/adder pairs are cascaded onto this structure as more subroutines are called. Note that logical register addresses above 31 are always applied to the first adder; and the output of the  $n$ th adder is the physical register address to be used in the  $n$ th subroutine. All registers and adders are only 7 bits wide because the largest physical register address that needs to be generated is 127.

- 11.12. Considering cacheing effects only, the average access time over both instruction and data accesses is a function of both cache hit rates and miss penalties (see Sections 5.6.2 and 5.6.3 for general expressions for average access time).

The hit rates in the 21264 L1 caches will be much higher than in the 21164 L1 caches because the 21264 caches are eight times larger. Therefore, the average access time for accesses that can be made on-chip will be larger in the 21164 because of the miss penalty in going to its on-chip L2 cache.

Next, we need to consider the effect on average access time of going to the off-chip caches in each system. The total on-chip cache capacity (112K bytes in the 21164 and 128K bytes in the 21264) is about the same in both the systems. Therefore, we can assume about the same hit rate for on-chip accesses; so the effect on average access time of the miss penalties in going to the off-chip caches will be about the same in each system.

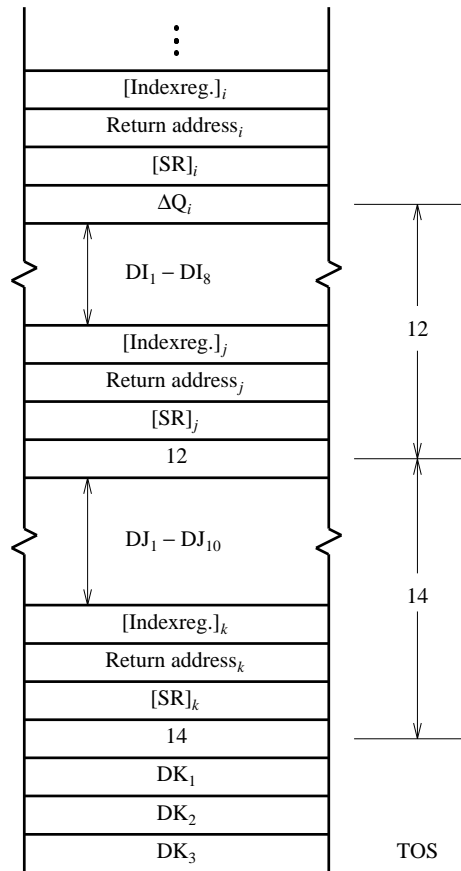
Finally, if the off-chip caches have about the same capacity, the effect on average access times of the miss penalties in going to the main DRAM memories will be about the same in each system.

The net result is that average access times in the 21264 should be shorter than in the 21164, leading to faster program execution, primarily because of the different arrangements of the on-chip caches.

- 11.13. HP3000 program:

|      |   |                                     |
|------|---|-------------------------------------|
| LOAD | A |                                     |
| LOAD | B |                                     |
| MPYM | C |                                     |
| LOAD | D |                                     |
| MPYM | E |                                     |
| ADD  |   |                                     |
| LOAD | F |                                     |
| MPYM | G |                                     |
| LOAD | H |                                     |
| MPYM | I |                                     |
| DIV  |   |                                     |
| DEL  |   | Combined with previous instruction. |
| ADD  |   |                                     |
| MPY  |   | Combined with previous instruction. |
| STOR | W |                                     |

- 11.14. Procedure<sub>*i*</sub> generates 8 words of data, Procedure<sub>*j*</sub> generates 10 words of data, and Procedure<sub>*k*</sub> generates 3 words of data. Then, the top words in the stack have the following contents:



11.15. HP3000 program:

```
LOAD  A
ADDM  B
LOAD  C
ADDM  D
MPY
LOAD  D
MPYM  E
ADD
STOR  W
```

ARM program:

```
LDR  R0,A
LDR  R1,B
ADD  R0,R0,R1
LDR  R1,C
LDR  R2,D
ADD  R1,R1,R2
LDR  R3,E
MUL  R2,R2,R3
MLA  R0,R0,R1,R2
STR  R0,W
```

68000 program (assume 16-bit operands):

```
MOVE  A,D0
ADD   B,D0
MOVE  C,D1
ADD   D,D1
MULS  D1,D0
MOVE  D,D1
MULS  E,D1
ADD   D1,D0
MOVE  D0,W
```

IA-32 program:

```
MOV    EAX,A
ADD    EAX,B
MOV    EBX,C
ADD    EBX,D
IMUL   EAX,EBX
MOV    EBX,D
IMUL   EBX,E
ADD    EAX,EBX
MOV    W,EAX
```

11.16. Four

11.17. Four and two

# Chapter 12 – Large Computer Systems

12.1. A possible program is:

```

LOOP  Move      0,STATUS
      Move      CURRENT,R1
      Move      R1,R2
      Move      R1,Rnet
      Shift_right Rnet
      Add        Rnet,R2      Add current value from left
      Move      R1,Rnet
      Shift_left  Rnet
      Add        Rnet,R2      Add current value from right
      Move      R1,Rnet
      Shift_up    Rnet
      Add        Rnet,R2      Add current value from below
      Move      R1,Rnet
      Shift_down  Rnet
      Add        Rnet,R2      Add current value from above
      Divide     5,R2          Average all five values
      Move      R2,CURRENT
      Subtract   R2,R1
      Absolute   R1
      Subtract   EPSILON,R1
      Skip_if $\geq$ 0
      Move      1,STATUS

      {Control processor ANDs all STATUS flags and exits LOOP if result is 1;
      otherwise, LOOP is repeated.}

END    LOOP
```

12.2. Assume that each bus has 64 address lines and 64 data lines. There are two cases to consider.

*i)* For uncached reads, each read with a split-transaction bus requires  $2T$ , consisting of  $1T$  to send the address to memory and  $1T$  to transfer the data to the processor. Using a conventional bus, it takes  $6T$  because of the  $4T$  delay in reading the contents of the memory. Therefore, 3 conventional buses would give approximately the same performance as the split-transaction bus.

*ii)* For cached reads, it is necessary to consider the size of the cache block. Assume that this size is 64 bytes; therefore, it takes 8 clock cycles to transfer an entire block over the bus.



Using a split-transaction bus it is possible to use all cycles to transfer either read requests (addresses) or data; therefore, it takes  $9T$  per read (not in consecutive clock cycles!). Using a conventional bus each read takes  $13T$  (consecutive clock cycles). Thus, 4 of these 13 cycles are wasted waiting for the memory response. This means that in this case also it would be necessary to use 3 conventional buses to obtain approximately the same performance.

- 12.3. The performance would not improve by a factor of 4, because some bus transactions involve uncached reads and writes. Since uncached accesses involve only one word of data, they use only one quarter of the 4-word wide bus. Of course, the overall performance would depend on the ratio of cached and uncached accesses.

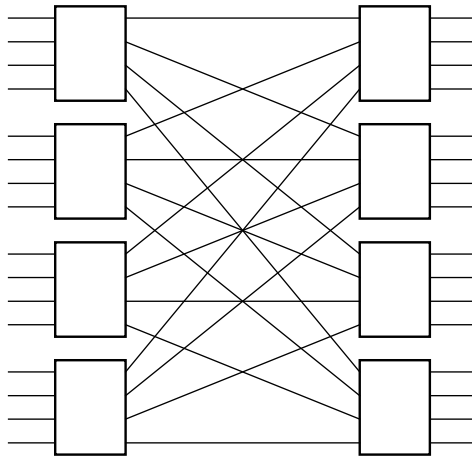
- 12.4. Assume  $n$  is a power of 2 because of the form of the shuffle network.

Crossbar cost =  $n^2$ .

Shuffle network cost =  $2(n/2)\log_2 n$ .

Solving for smallest  $n$  satisfying  $n^2 \geq 5[2(n/2)\log_2 n]$  where  $n$  is a power of 2, gives  $n \geq 5\log_2 n$ . At  $n = 16$ , inequality is not satisfied. At  $n = 32$ , inequality is satisfied. Therefore, the smallest  $n$  is 32.

- 12.5. The network is



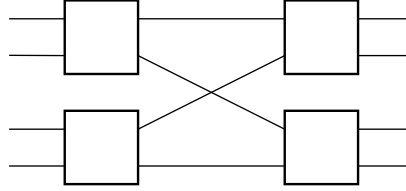
Note that the definition of the shuffle pattern must be generalized in such a way that for each source input there is a path (in fact, exactly one path) to each destination output.

Cost of network built from  $2 \times 2$  switches is  $(n/2)\log_2 n$ .

Cost of network built from  $4 \times 4$  switches is  $4(n/4)\log_4 n = n(\log_2 n / \log_2 4) = (n/2)\log_2 n$ .

Therefore, the cost of the two types of networks is the same.

Blocking probability: The  $4 \times 4$  switch is a nonblocking crossbar, and can be built from  $2 \times 2$  switches as



But this is a blocking network. Therefore, the blocking probability of a large network built from  $4 \times 4$  switches is lower than one built from  $2 \times 2$  switches.

#### 12.6. Program structure:

Sequential segment  $S_1$  ( $k$  time units)  
 PAR segment  $P_1$  (1 time unit)  
 Sequential segment  $S_2$  ( $k$  time units)  
 PAR segment  $P_2$  (1 time unit)  
 Sequential segment  $S_3$  ( $k$  time units)

$$T_1 = 3k + 2k$$

$$T_n = 3k + 2\lceil(k/n)\rceil$$

$$\text{Speedup} = (5k)/(3k + 2\lceil(k/n)\rceil)$$

Limiting value for speedup is  $5/3$ . This shows that the sequential segments of a program severely limit the speedup when the sequential segments take about the same time to execute as the time taken to execute the PAR segments on a single processor.

- 12.7. The  $n$ -dimensional hypercube is symmetric with respect to any node. The distance between nodes  $x$  and  $y$  is the number of bit positions that are different in their binary addresses. The number of nodes that are  $k$  hops away from any particular node is  $\binom{n}{k}$ . Therefore, the average distance a message travels is

$$\left[ \sum_{k=1}^n k \cdot \binom{n}{k} \right] / (2^n - 1)$$

which simplifies to  $[2^{n-1} \cdot n] / (2^n - 1)$ , and is less than  $(1 + n)/2$ , as can be verified by trying a few values. For large  $n$ , the average distance approaches  $n/2$ .

- 12.8. When a Test-and-Set instruction “fails,” that is, when the lock was already set, the task should call the operating system to have its task name queued and to allow some other task to execute. When the task holding the lock wishes to release the lock (set it to 0), the task calls the operating system to do so, and then the operating system dequeues and runs one of the waiting tasks which is then

the one owning the lock. If no task is waiting, the lock is cleared ( $= 0$ ) to the free state.

- 12.9. The details of how either invalidation or updating can be implemented are described in Section 12.6.2, and the advantages/disadvantages of the two techniques can be deduced directly from that discussion. In general, it would seem that invalidation and write-back of dirty variables results in less bus traffic and eliminates potentially wasted cache updating operations. However, cache hit rates may be lowered by using this strategy. Updating associated with a write through policy may lead to higher hit rates and may be simpler to implement, but may cause unacceptably high bus traffic and wasted update operations. The details of how reads and writes on shared cached blocks (lines) are normally interleaved from distinct processors in some class of applications will actually determine which coherence strategy is most appropriate.
- 12.10. No. If coherence controls are not used, a shared variable in cache B may not get updated/invalidated when it is written to in cache A while A's processor has mutually exclusive access. Later, when B's processor acquires mutually exclusive access, the variable will be incorrect.
- 12.11. In Figure 12.18, both threads continuously write the same shared variable *dot\_product*; hence, this is done serially. In Figure 12.19, each thread updates its local variable *local\_dot\_product*, which is done in parallel. Therefore, if very large vectors are used (so that the actual computation of the dot product dominates the processing time), the program in Figure 12.19 may give almost twice as good performance as the program in Figure 12.18.
- 12.12. It is only necessary to create 3 new threads (rather than just one in Figure 12.19), and assign processing of one quarter of each vector to each thread.
- 12.13. The only significant modification is for the program with  $id = 0$  to send one quarter of each vector to programs with  $id = 1, 2, 3$ . Having completed the dot-product computation, each program with  $id > 0$  sends the result to the program with  $id = 0$ , which then prints the final result.
- 12.14. Overhead in creating a thread is the most significant consideration. Other overhead is encountered in the lock and barrier mechanisms. Assume that the thread overhead is 300 times greater than the execution time of the statement that computes the new value of the dot product for a given value of  $k$ . Also, assume that the overhead for lock and barrier mechanisms is only 10 times greater.  
Then, as a rough approximation, the vectors must have at least  $320 \times 2 = 640$  elements before any speedup will be achieved.
- 12.15. The dominant factor in message passing is the overhead of sending and receiving messages. Assume that the overhead of either sending or receiving a message is 1000 times greater than the execution time of the statement that computes the new value of the dot product for a given value of  $k$ . Then, since there are 3

send and 3 receive messages involved, the vectors will have to have at least  $1000 \times 6 = 6000$  elements before any speedup is achieved.

Note that we have assumed that the overhead of 1000 is independent of the size of the message – as a first order approximation.

- 12.16. The shared-memory multiprocessor can emulate the message-passing multicomputer easier than the other way around. The act of message-passing can be implemented by the transfer of (message) buffer pointers or complete (message) buffers between the two communicating processes that otherwise only operate in their own assigned area of main memory. A multicomputer system can emulate a multiprocessor by considering the aggregate of all of the local memories of the individual computers as the shared memory of the multiprocessor. Access from a computer to a nonlocal component of the shared memory can be facilitated by passing messages between the two computers involved. This is a cumbersome and slow process.
- 12.17. The situation described is possible. Consider stations A, B, and C, situated at the left end, middle, and right end of the bus, respectively. Station A starts to send a message packet of  $0.25\tau$  duration to destination station B at time  $t_0$ . The packet is observed and copied into station B during the interval  $[t_0 + 0.5\tau, t_0 + 0.75\tau]$ . Just before  $t_0 + \tau$ , station C begins to transmit a packet to some other station. It immediately collides with A's packet, and the garbled signal arrives back at station A just before  $t_0 + 2\tau$ .
- 12.18. (a) The F/E bit is tested. If it is 1 (denoting "full"), then the contents of BOXLOC are loaded into register R0, F/E is set to 0 (denoting "empty"), and execution continues with the next sequential instruction. Otherwise (i.e., for  $[F/E] = 0$ ), no operations are performed and execution control is passed to the instruction at location WAITREC.

(b) In the multiprocessor system with the mailbox memory, each one-word message is sent from  $T_1$  to  $T_2$  by using the single instructions:

SEND   PUT   R0,BOXLOC,SEND            (1)

and

REC     GET   R0,BOXLOC,REC            (2)

in tasks  $T_1$  and  $T_2$ , respectively, assuming that  $[F/E] = 0$  initially.

In the system without the mailbox memory, replace (1) in task  $T_1$  with the sequence:

|       |       |        |
|-------|-------|--------|
| WLOCK | TAS.B | WRITE  |
|       | BMI   | WLOCK  |
|       | MOV.W | R0,LOC |
|       | CLR.B | READ   |

and replace (2) in task  $T_2$  with the sequence:

|       |       |         |
|-------|-------|---------|
| RLOCK | TAS.B | READ    |
|       | BMI   | RLOCK   |
|       | MOV.W | LOCK,R0 |
|       | CLR.B | WRITE   |

Let the notation  $V(7)$  stand for bit  $b_7$  of byte location  $V$ . Ordinary word location  $LOC$  represents the data field of mailbox location  $BOXLOC$ , and the combination of  $WRITE(7)$  and  $READ(7)$  represents the F/E bit associated with  $BOXLOC$ .

In particular,  $[WRITE(7)] = 0$  means that  $LOC$  is empty, and  $[READ(7)] = 0$  means that  $LOC$  is full.

Initially, when  $LOC$  is empty, the settings must be  $[WRITE(7)] = 0$  and  $[READ(7)] = 1$ . Note that when the instruction  $MOV.W$  is being executed in either task, we have  $[WRITE(7)] = [READ(7)] = 1$ , indicating that  $LOC$  is either being filled or emptied. Also note that it is never the case that  $[WRITE(7)] = [READ(7)] = 0$ . This solution works correctly for the general case where a number of tasks pass data through  $LOC$ .

For the case suggested in the problem, with a single task  $T_1$  and a single task  $T_2$ , the following sequences are sufficient. In  $T_1$ , use:

|       |       |         |
|-------|-------|---------|
| TESTW | TST.B | FULL    |
|       | BNE   | TESTW   |
|       | MOV.W | R0,LOC  |
|       | MOV.B | #1,FULL |

In  $T_2$  use:

|       |       |        |
|-------|-------|--------|
| TESTR | TST.B | FULL   |
|       | BEQ   | TESTR  |
|       | MOV.W | LOC,R0 |
|       | CLR.B | FULL   |

In this case,  $FULL$  plays the role of the F/E bit of  $BOXLOC$  (with  $[FULL] = 0$  initially), and the  $TAS$  instruction is not needed.