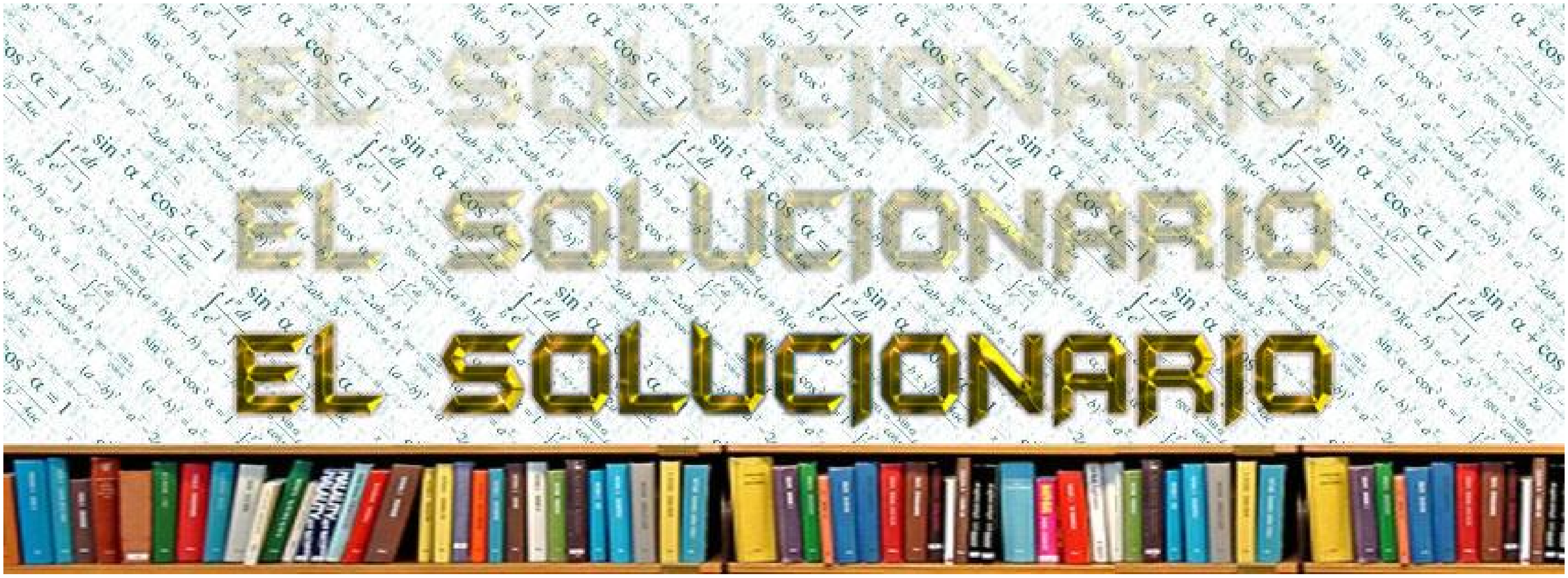
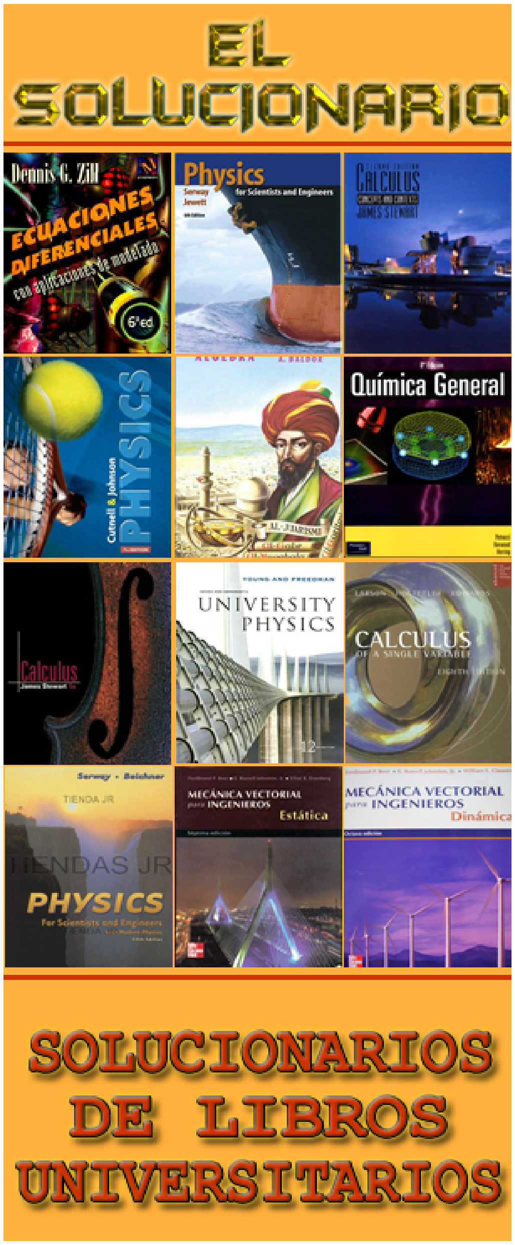

EL SOLUCIONARIO
EL SOLUCIONARIO
EL SOLUCIONARIO


EL SOLUCIONARIO
SOLUCIONARIOS DE LIBROS UNIVERSITARIOS

CONTENTS

Chapter 3

ARM, MOTOROLA, AND INTEL INSTRUCTION SETS 103

Chapter 4

INPUT/OUTPUT ORGANIZATION 203

Chapter 5
THE MEMORY SYSTEM 291

Chapter 6
ARITHMETIC 367

Chapter 7

BASIC PROCESSING UNIT 411

Chapter 8

PIPELINING 453

Chapter 9

EMBEDDED SYSTEMS 511

Chapter 10
COMPUTER PERIPHERALS 553

Chapter 11
PROCESSOR FAMILIES 577

Chapter 12
LARGE COMPUTER SYSTEMS 617

APPENDIX A: LOGIC CIRCUITS 661

APPENDIX B: ARM INSTRUCTION SET 733

APPENDIX C: MOTOROLA 68000 INSTRUCTION SET 751

APPENDIX D: INTEL IA-32 INSTRUCTION SET 769

APPENDIX E: CHARACTER CODES AND NUMBER CONVERSION 789

INDEX 795

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

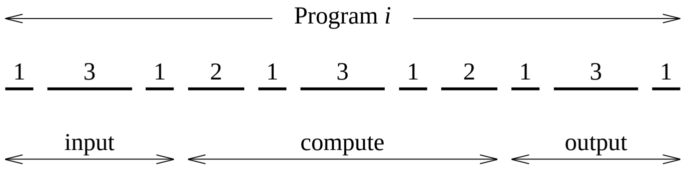
1.3. (a)

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

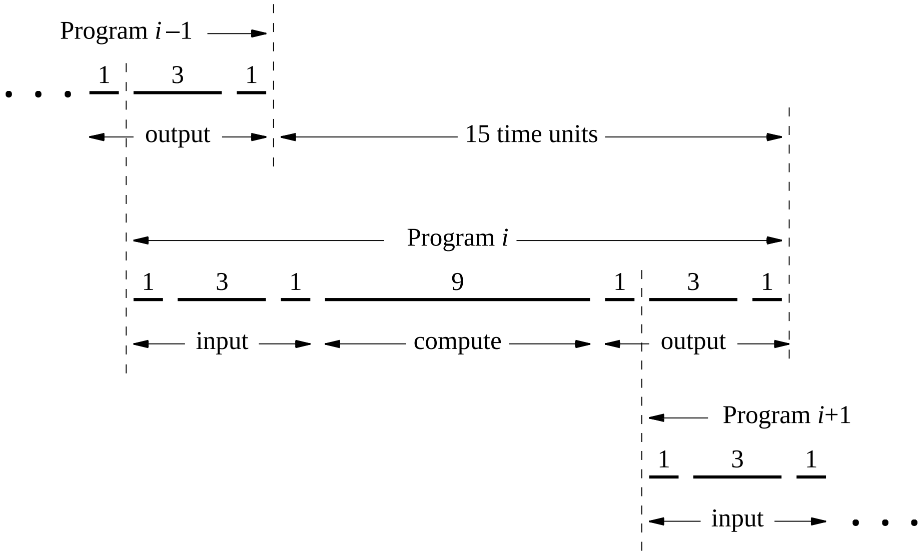
(b) Yes;

```
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 $\mathrm{T_R} = (\mathrm{N_R} \times \mathrm{S_R}) \,/\, \mathrm{R_R}$ and $\mathrm{T_C} = (\mathrm{N_C} \times \mathrm{S_C}) \,/\, \mathrm{R_C}$ be execution times on the RISC and CISC processors, respectively. Equating execution times and clock rates, we have

$$1.2\,\mathrm{N_R} = 1.5\,\mathrm{N_C}$$

Then

$$\mathrm{N_C} \,/\, \mathrm{N_R} = 1.2 \,/\, 1.5 = 0.8$$

Therefore, the largest allowable value for $\mathrm{N_C}$ is 80% of $\mathrm{N_R}$.

(b) In this case

$$1.2\,\mathrm{N_R}\,/\,1.15 = 1.5\,\mathrm{N_C}\,/\,1.00$$

Then

$$\mathrm{N_C}\,/\,\mathrm{N_R} = 1.2\,/\,(1.15 \times 1.5) = 0.696$$

Therefore, the largest allowable value for $\mathrm{N_C}$ is 69.6% of $\mathrm{N_R}$.

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*)

$$\begin{array}{lrlrlr} (a) & 00101 & (b) & 00111 & (c) & 10010 \\ & \underline{+\ 01010} & & \underline{+\ 01101} & & \underline{+\ 01011} \\ & 01111 & & 10100 & & 11101 \\ & \text{no overflow} & & \text{overflow} & & \text{no overflow} \end{array}$$

$$\begin{array}{lrlrlr} (d) & 11011 & (e) & 11101 & (f) & 10110 \\ & \underline{+\ 00111} & & \underline{+\ 11000} & & \underline{+\ 10011} \\ & 00010 & & 10101 & & 01001 \\ & \text{no overflow} & & \text{no overflow} & & \text{overflow} \end{array}$$

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

$$\begin{array}{lrlrlr} (a) & 00101 & (b) & 00111 & (c) & 10010 \\ & \underline{+\ 10110} & & \underline{+\ 10011} & & \underline{+\ 10101} \\ & 11011 & & 11010 & & 00111 \\ & \text{no overflow} & & \text{no overflow} & & \text{overflow} \end{array}$$

$$\begin{array}{lrlrlr} (d) & 11011 & (e) & 11101 & (f) & 10110 \\ & \underline{+\ 11001} & & \underline{+\ 01000} & & \underline{+\ 01101} \\ & 10100 & & 00101 & & 00011 \\ & \text{no overflow} & & \text{no overflow} & & \text{no overflow} \end{array}$$

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 736F6EXX. 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 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 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.
         Move        J,R10        Initialize outer loop counter R10 to j.
OUTER    Move        N,R11        Initialize inner loop counter R11 to n.
         Clear       R2           Clear index register R2 to zero.
         Clear       R0           Clear sum register R0 to zero.
INNER    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
                                         ________

        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)
           Multiply    #4,R2            to A(0,x) and A(0,y)
                                        placed in R1 and R2.
LOOP       Move        (R0,R1),R5     Add corresponding
           Add         R5,(R0,R2)       column elements.
           Add         R4,R1          Increment column element
           Add         R4,R2            pointers by Stride value.
           Decrement   R3             Repeat until all
           Branch>0    LOOP             elements are added.
           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*)

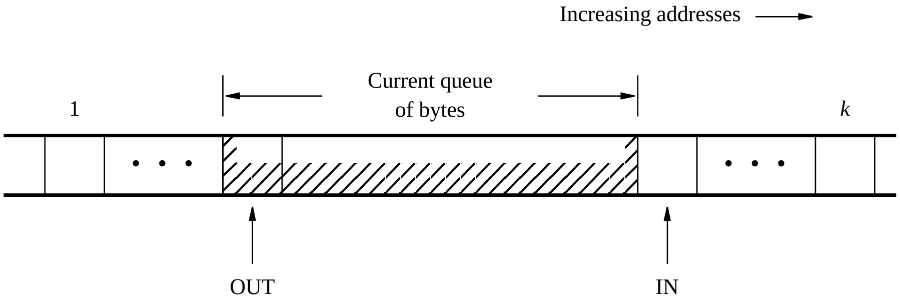
```
Move   16(R5),R3
```

(*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) Modulo k = [OUT]. If this is done, the queue is empty only when [IN] = [OUT].

(*e*) Append operation:

- LOC ← [IN]
- IN ← ([IN] + 1) 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 ← ([OUT] + 1) 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≥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≥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≤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≠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≠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≠0   CHECKHEAD                return with RIDNUM unchanged.
             Return
CHECKHEAD    Compare    (RHEAD), RIDNUM        Check if Head record
             Branch≠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≠0   CHECKNEXT                return with IDNUM unchanged.
             Return
CHECKNEXT    Compare    (RNEXT), RIDNUM        Check if next record is
             Branch≠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 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.

```
          Move        #10,R0       Use R0 as counter
          Move        #LOC,R1      Use R1 as pointer
Next      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≥0    Letters      Branch if [R3] ≥ 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.

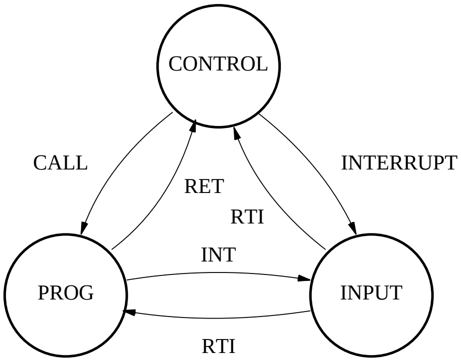
4.6. (*a*) Interrupts should be enabled, except when C is being serviced. The nesting rules can be enforced by manipulating the interrupt-enable flags in the interfaces of A and B.

(*b*) A and B should be connected to INTR_2, and C to 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, PNTRn, 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        Move         #20,R1             Use R1 as device counter, i
LOOP        Move         DEVICES(R1),R2     Get address of device i
            BitTest      #0,2(R2)           Test input status of a device
            Branch≠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:

Case *a*: Time $= 100 \times 800 \times 10^{-9}$ ns $= 80\ \mu$s
Case *b*: Time $= 20 \times r \times 200 \times 10^{-9} \times 100 = 400r$ 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 μs 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.

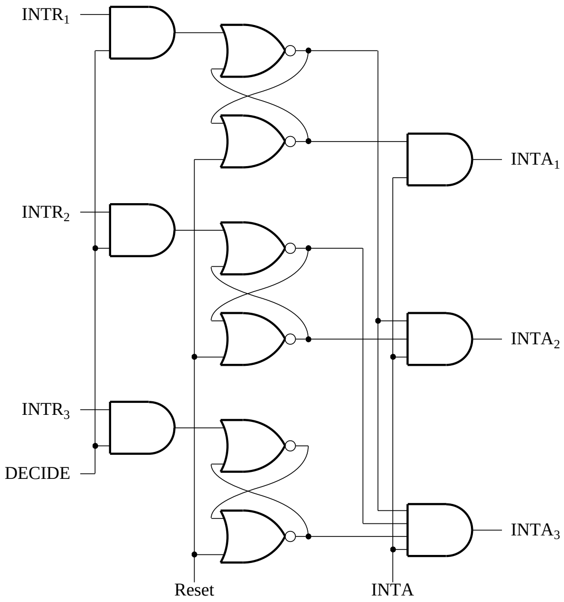
(*c*) 36 for completion of current instruction plus 6 for interrupt handling, for a total of 42.

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

(*b*) See logic equations in part *a*.

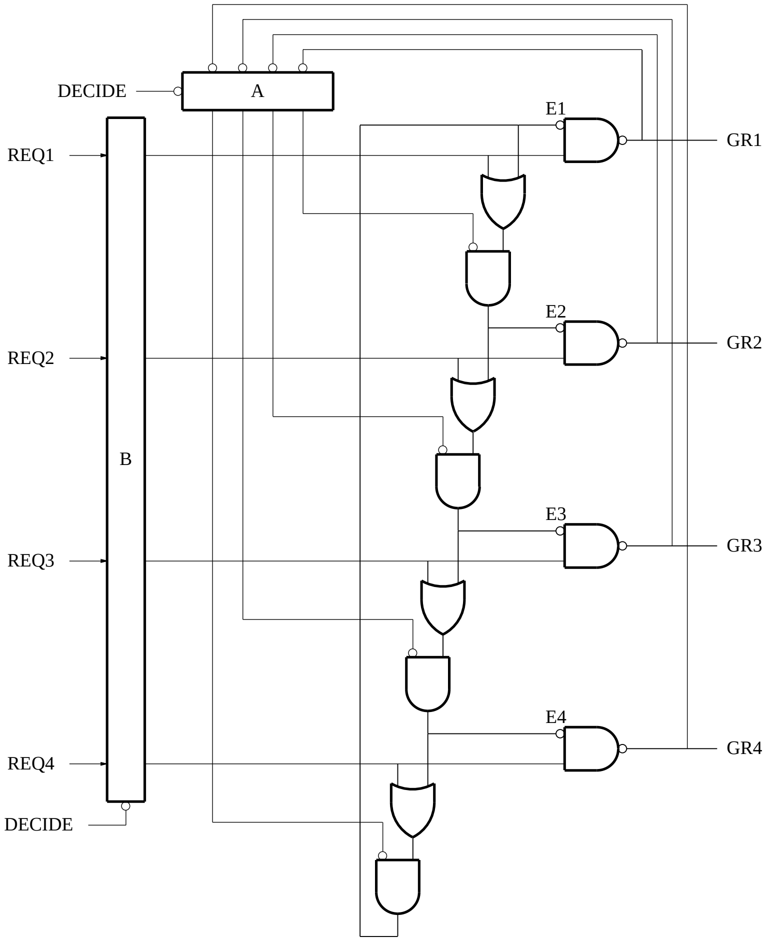
(*c*) Yes.

(*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_2	IPL_1	IPL_0
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:

$$\mathrm{IPL}_2 = \mathrm{q}_4 + \mathrm{q}_5 + \mathrm{q}_6 + \mathrm{q}_7$$

$$\mathrm{IPL}_1 = \mathrm{q}_6 + \mathrm{q}_7 + \overline{\mathrm{IPL}_2}(\mathrm{q}_2 + \mathrm{q}_3)$$

$$\mathrm{IPL}_0 = \mathrm{q}_7 + \mathrm{q}_5 \cdot \mathrm{q}_6 + \overline{\mathrm{IPL}_2}(\mathrm{q}_3 + \mathrm{q}_1 \cdot \mathrm{q}_2)$$

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_{16}) 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.

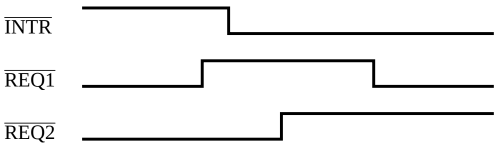
```
LDR   R2,[R14,#-4]        Get SWI instruction
BIC   R2,R2,#&FFFFFF00    Clear high-order bits
```

(*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.

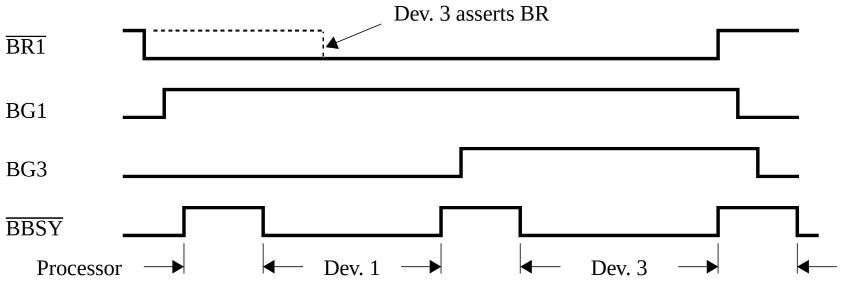
```
ADR   R3,EntryTable          Get the table's address
LDR   R15,[R3,R2,LSL #2]     Load starting address of routine
```

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.



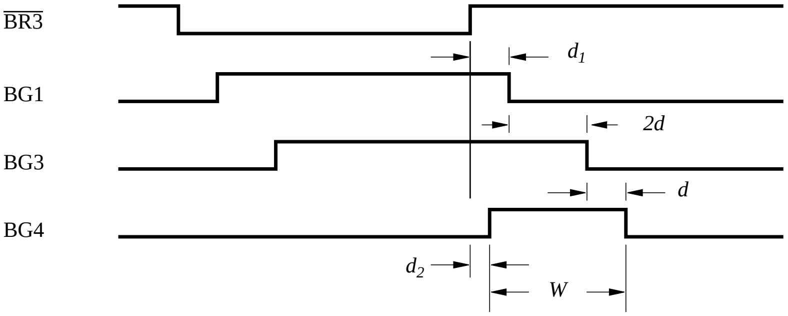
4.21. Assume that when BR becomes active, the processor asserts BG1 and keeps it asserted until BR is negated.



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.

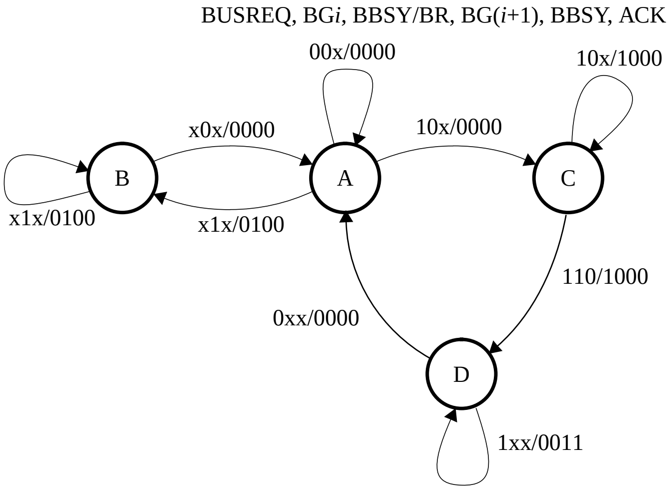
4.23. *For better clarity, change* BR *to* $\overline{\text{BR}}$ *and use an inverter with delay* d_1 *to generate* BG1.



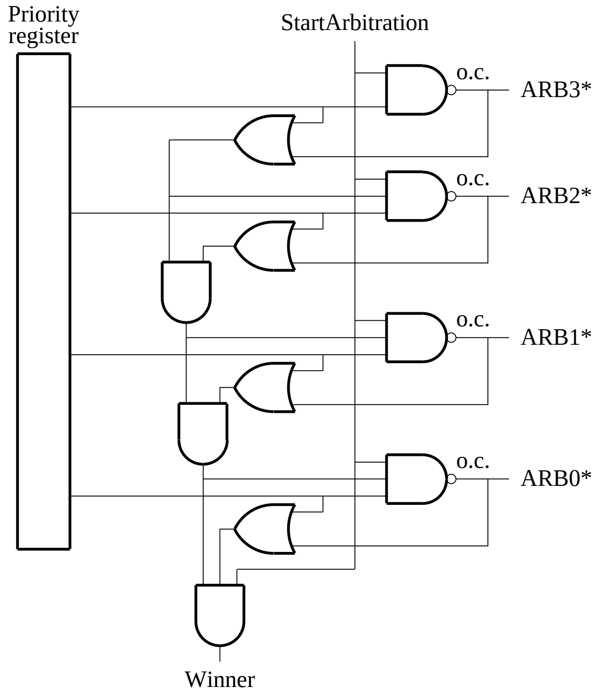
Assuming device 3 asserts BG4 shortly after it drops the bus request (delay d_2), a spurious pulse of width $W = d_1 + 3d - d_2$ will appear on BG4.

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 4.22*b* above is observed (state B).



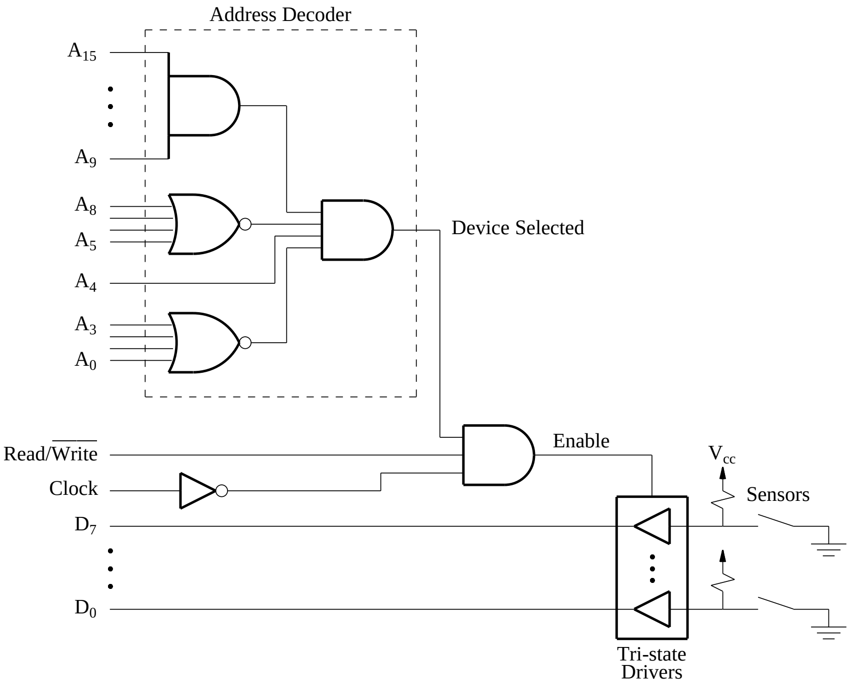
4.26. The priority register in the circuit below contains 1111 for the highest priority device and 0000 for the lowest.



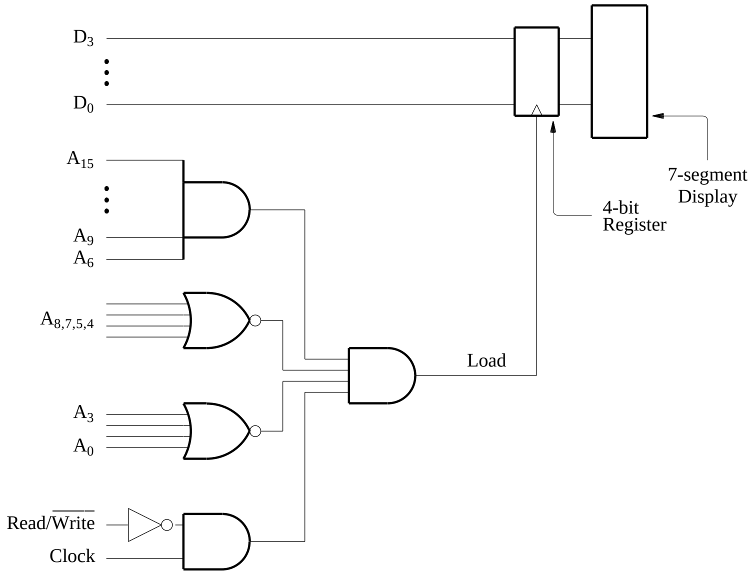
4.27. A larger distance means longer delay for the signals traveling between the processor and the input device. Primarily, this means that $t_2 - t_1$, $t_3 - t_2$ and $t_5 - t_3$ will increase. Since longer distances may also mean larger skew, the intervals $t_1 - t_0$ and $t_4 - t_3$ may have to be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

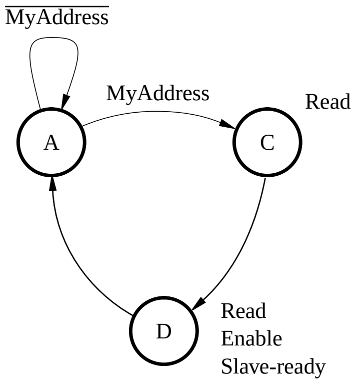


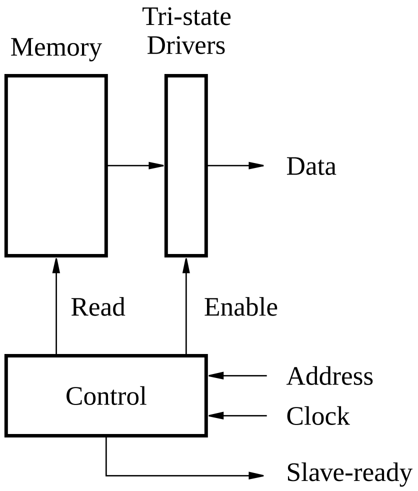
4.29. Assume that the display has the bus address FE40. The circuit below sets the Load signal to 0 during the second half of the write cycle. The rising edge at the end of the clock period will load the data into the display register.



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 $\overline{\text{INTR}} = \overline{\text{SIN} \cdot \text{IntEn}}$.

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

Chapter 5 – The Memory System

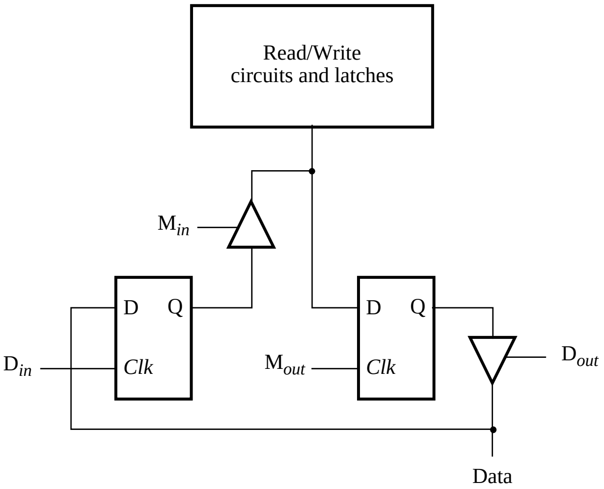
5.1. The block diagram is essentially the same as in Figure 5.10, except that 16 rows (of four 512×8 chips) are needed. Address lines A_{18-0} are connected to all chips. Address lines A_{22-19} are connected to a 4-bit decoder to select one of the 16 rows.

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.

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

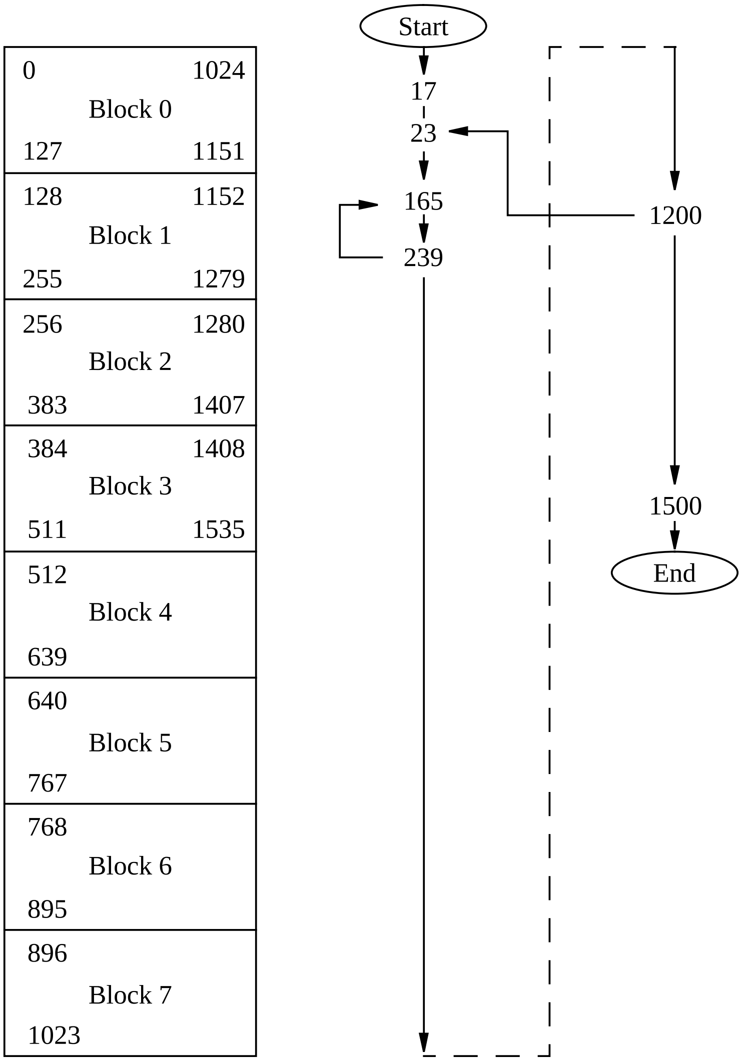
$$\text{Latency} = \frac{5}{(133 \times 10^6)} = 0.038 \times 10^{-6} \text{ s} = 38 \text{ ns}$$

(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 : $\underbrace{0,1,2,3,4,5,6,7,0,1,}_{\text{Pass 1}}\ \underbrace{0,1,0,1,}_{\text{Pass 2}}\ 0,1,\ldots,0,1,\ \underbrace{0,1,0,1,}_{\text{Pass 9}}\ \underbrace{0,1,0,1,2,3}_{\text{Pass 10}}$

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:

$$\begin{aligned}
\text{Outer loop} - \text{inner loop} &= [(1200-22)-(239-164)]10 \times 1\tau = 11,030\,\tau \\
\text{Inner loop} &= (239-164)200 \times 1\,\tau = 15,000\,\tau \\
\text{End section of program} &= 1500-1200 = 300 \times 1\,\tau \\
\text{Total execution time} &= 87,770\,\tau
\end{aligned}$$

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 005D Add 005D Dec	4	0	40 τ
2	005E BNE 005D Add 005D Dec	2	2	22 τ
3	005E BNE 00AA 10D7 005D Add 005D Dec	1	3	13 τ
Total		7	5	75 τ

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 τ.

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, $\cdots$, 4351 occupy blocks 0 to 67 in the main memory (MM). After blocks 0, 1, 2, $\cdots$, 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, $\cdots$, 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

	Contents of data cache after pass:					
Block position	*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

	Contents of data cache after pass:			
Block position	*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

	Block position	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

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	[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]

Hit rate = 33/48 = 0.69

(b) Associative-mapped cache

	Contents of data cache after:			
Block position	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

		Contents of data cache after:			
	Block position	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

	Contents of data cache after:			
Block position	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

	Contents of data cache after:			
Block position	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:			
	Block position	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

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

(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 1024×40 ms = 41 s

Total time for (b) is 2048×40 ms = 82 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

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

Therefore, average access and read time is

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

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

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

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.

$$\begin{array}{rrrrrr}
010110 & (+22) & 101011 & (-21) & 111111 & (-1) \\
+\ 001001 & +\ (+9) & +\ 100101 & +\ (-27) & +\ 000111 & +\ (+7) \\
\hline
011111 & (+31) & 010000 & (-48) & 000110 & (+6) \\
 & & \text{overflow} & & &
\end{array}$$

$$\begin{array}{rrrrrr}
011001 & (+25) & 110111 & (-9) & 010101 & (+21) \\
+\ 010000 & +\ (+16) & +\ 111001 & +\ (-7) & +\ 101011 & +\ (-21) \\
\hline
101001 & (+41) & 110000 & (-16) & 000000 & (0) \\
\text{overflow} & & & & &
\end{array}$$

$$\begin{array}{rrr}
010110 & (+22) & 010110 \\
-\ 011111 & -\ (+31) & +\ 100001 \\
\hline
 & (-9) & 110111
\end{array}$$

$$\begin{array}{rrr}
111110 & (-2) & 111110 \\
-\ 100101 & -\ (-27) & +\ 011011 \\
\hline
 & (+25) & 011001
\end{array}$$

$$\begin{array}{rrr}
100001 & (-31) & 100001 \\
-\ 011101 & -\ (+29) & +\ 100011 \\
\hline
 & (-60) & 000100 \\
 & & \text{overflow}
\end{array}$$

$$\begin{array}{rrr}
111111 & (-1) & 111111 \\
-\ 000111 & -\ (+7) & +\ 111001 \\
\hline
 & (-8) & 111000
\end{array}$$

$$\begin{array}{rrr}
000111 & (+7) & 000111 \\
-\ 111000 & -\ (-8) & +\ 001000 \\
\hline
 & (+15) & 001111
\end{array}$$

$$\begin{array}{rrr}
011010 & (+26) & 011010 \\
-\ 100010 & -\ (-30) & +\ 011110 \\
\hline
 & (+56) & 111000 \\
 & & \text{overflow}
\end{array}$$

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	Ternary	Ternary
Sign-and-magnitude	Sign-and-magnitude	3’s-complement
56	+2002	002002
−37	−1101	221122
122	11112	011112
−123	−11120	211110

Addition operations:

$$\begin{array}{r} 002002 \\ +\ 221122 \\ \hline 000201 \end{array} \qquad \begin{array}{r} 002002 \\ +\ 011112 \\ \hline 020121 \end{array} \qquad \begin{array}{r} 002002 \\ +\ 211110 \\ \hline 220112 \end{array}$$

$$\begin{array}{r} 221122 \\ +\ 011112 \\ \hline 010011 \end{array} \qquad \begin{array}{r} 221122 \\ +\ 211110 \\ \hline 210002 \end{array} \qquad \begin{array}{r} 011112 \\ +\ 211110 \\ \hline 222222 \end{array}$$

Subtraction operations:

$$\begin{array}{r} 002002 \\ -\ 221122 \\ \hline \\ \end{array} \qquad \begin{array}{r} 002002 \\ +\ 001101 \\ \hline 010110 \end{array}$$

$$\begin{array}{r} 002002 \\ -\ 011112 \\ \hline \\ \end{array} \qquad \begin{array}{r} 002002 \\ +\ 211111 \\ \hline 220120 \end{array}$$

$$\begin{array}{r} 002002 \\ -\ 211110 \\ \hline \\ \end{array} \qquad \begin{array}{r} 002002 \\ +\ 011120 \\ \hline 020122 \end{array}$$

$$\begin{array}{r} 221122 \\ -\ 011112 \\ \hline \\ \end{array} \qquad \begin{array}{r} 221122 \\ +\ 211111 \\ \hline 210010 \end{array}$$

$$\begin{array}{r} 221122 \\ -\ 211110 \\ \hline \\ \end{array} \qquad \begin{array}{r} 221122 \\ +\ 011120 \\ \hline 010012 \end{array}$$

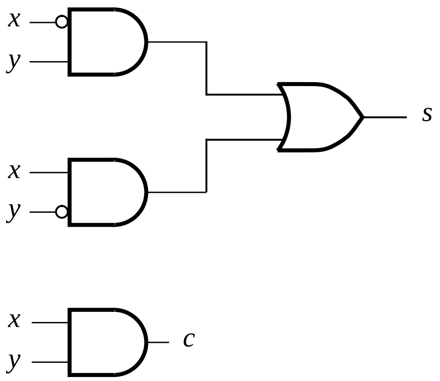
$$\begin{array}{r} 011112 \\ -\ 211110 \\ \hline \\ \\ \end{array} \qquad \begin{array}{r} 011112 \\ +\ 011120 \\ \hline 100002 \\ \text{overflow} \end{array}$$

6.5. (*a*)

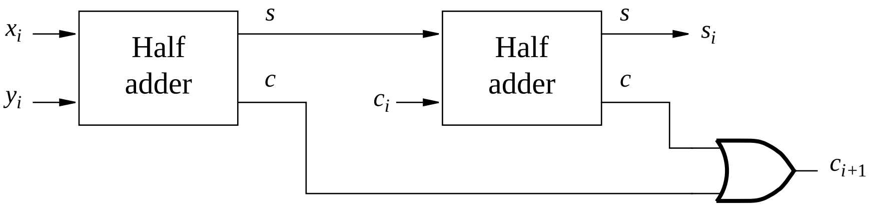
x	y	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$s = x \oplus y$

$c = x\,y$



(*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 6.2*a* 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:

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

IA-32 Program:

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

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] = D0.
          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 D1 into R3.
          MLA     R0,R3,R4,R0         Add 10D1 into R0.
          AND     R3,R2,R1,LSR #8     Get D2 into R3.
          MLA     R0,R3,R5,R0         Add 100D2 into R0.
          AND     R3,R2,R1,LSR #12    Get D3 into R3.
          MLA     R0,R3,R6,R0         Add 1000D3 into Ro.
          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] = D0.
          SHR    ESI,4
          MOV    EAX,ESI
          AND    EAX,FH
          MUL    [EBP − 4]
          ADD    EDX,EAX          [EDX] = binary of D1D0.
          SHR    ESI,4
          MOV    EAX,ESI
          AND    EAX,FH
          MUL    [EBP − 8]
          ADD    EDX,EAX          [EDX] = binary of D2D1D0.
          SHR    ESI,4
          MOV    EAX,ESI
          AND    EAX,FH
          MUL    [EBP − 12]
          ADD    EDX,EAX          [EDX] = binary of D3D2D1D0.
          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 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.
          MOVE.B    #3,D2         Load counter D3.
LOOP      MULU.W    #10,D0        Multiply accumulated
                                   value in D0 by 10.
          ASL.L     #4,D1         Bring next $D_i$ digit
          SWAP.W    D1             into low half of D1.
          ADD.W     D1,D0         Add into accumulated
                                   value in D0.
          CLR.W     D1            Clear out current
          SWAP.W    D1             digit and bring
                                   remaining digits into
                                   low half of 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)

$$\begin{array}{rcr} 0101 & & 5 \\ +\,0110 & & +\,6 \\ \hline 1011 & > 10_{10} & 11 \end{array}$$

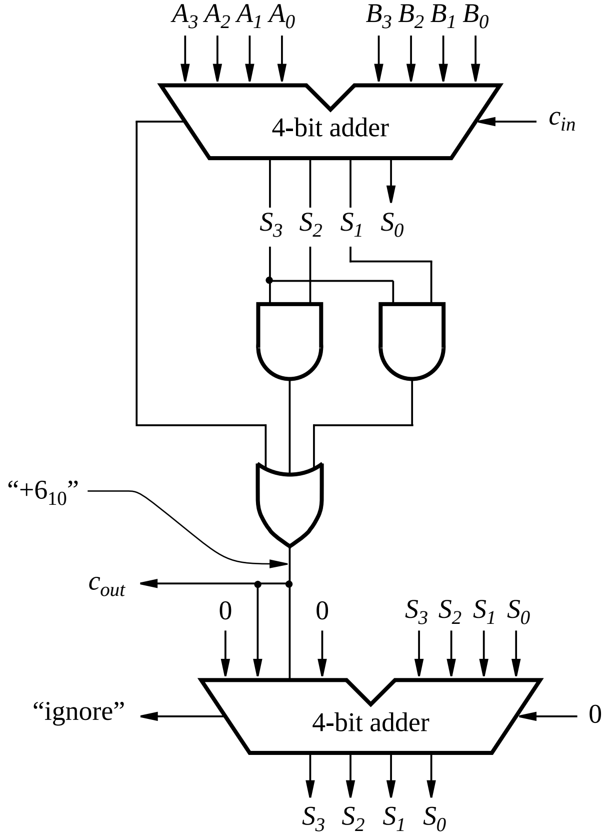
$$\begin{array}{r} +\,0110 \\ \hline 0001 \end{array}$$

output carry = 1

(2)

$$\begin{array}{rcr} 0011 & & 3 \\ +\,0100 & & +\,4 \\ \hline 0111 & < 10_{10} & 7 \end{array}$$

(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.)

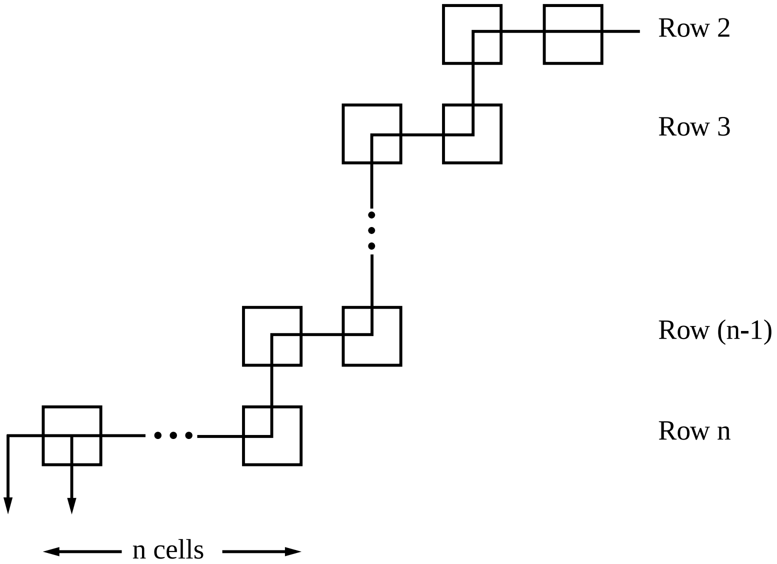
(b) The inputs G_i^{II} and P_i^{II} to the additional logic are produced after 5 gate delays, the same as the delay for c_{16} in Figure 6.5. Then all outputs from the additional logic, including c_{64}, are produced 2 gate delays later, for a total of 7 gate delays. The carry input c_{48} to the last 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:

```
   B  =  00101         ( 5)
  ×A  =  10101        × (21)
  ----   -----        ------
         00101         (105)
             0
       00101
     001010
     ---------
     001101001         (105)
```

```
               100           4
     00101 √10101        5 √21
             101            20
             ---            --
             00001           1
```

6.14. The multiplication and division charts are:

A × B :

	M		
	$\boxed{00101}$		
$\boxed{0}$	$\boxed{00000}$	$\boxed{10101}$	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 :

	A	Q	
	$\boxed{000000}$	$\boxed{10101}$	
	A	Q	Initial configuration
	$\boxed{000101}$		
	M		
shift	000001	0 1 0 1 □	
subtract	111011		1st cycle
	111100	0 1 0 1 $\boxed{0}$	
shift	111000	1 0 1 $\boxed{0}$ □	
add	000101		2nd cycle
	111101	1 0 1 $\boxed{0}$ $\boxed{0}$	
shift	111011	0 1 $\boxed{0}$ $\boxed{0}$ □	
add	000101		3rd cycle
	000000	0 1 $\boxed{0}$ $\boxed{0}$ $\boxed{1}$	
shift	000000	1 $\boxed{0}$ $\boxed{0}$ $\boxed{1}$ □	
subtract	111011		4th cycle
	111011	1 $\boxed{0}$ $\boxed{0}$ $\boxed{1}$ $\boxed{0}$	
shift	110111	$\boxed{0}$ $\boxed{0}$ $\boxed{1}$ $\boxed{0}$ □	
add	000101		5th cycle
	111100	$\boxed{0}$ $\boxed{0}$ $\boxed{1}$ $\boxed{0}$ $\boxed{0}$	
add	000101	quotient	
	000001		
	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:       . . .
```

6.17. The multiplication answers are:

(*a*)

$$\begin{array}{r} 010111 \\ \times\, 110110 \\ \hline \end{array} \qquad \begin{array}{r} +23 \\ \times\ -10 \\ \hline -230 \end{array}$$

$$\begin{array}{r cccccccccccc}
 & & & & & & & 0 & 1 & 0 & 1 & 1 & 1 \\
 & & & & & & \times & 0 & -1 & +1 & 0 & -1 & 0 \\ \cline{8-13}
 & & & & & & & & & & & & 0 \\
 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & \\
 & & & & & & & & & & 0 & & \\
\text{sign extension} & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & & & \\
 & 1_1 & 1_1 & 1_1 & 0_2 & 1_1 & 0_1 & 0_1 & 1 & & & & \\ \cline{2-13}
 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0
\end{array}$$

(*b*)

$$\begin{array}{r} 110011 \\ \times\, 101100 \\ \hline \end{array} \qquad \begin{array}{r} -13 \\ \times\ -20 \\ \hline 260 \end{array}$$

$$\begin{array}{r cccccccccccc}
 & & & & & & & 1 & 1 & 0 & 0 & 1 & 1 \\
 & & & & & & \times & -1 & +1 & 0 & -1 & 0 & 0 \\ \cline{8-13}
 & & & & & & & & & & & & 0 \\
 & & & & & & & & & & & 0 & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & & \\
 & & & & & & & & & 0 & & & \\
\text{sign extension} & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & & & & \\
 & 0_1 & 0_1 & 0_1 & 1_1 & 1_1 & 0_2 & 1_1 & & & & & \\ \cline{2-13}
 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}$$

(*c*)

$$\begin{array}{r} 110101 \\ \times\, 011011 \\ \hline \end{array} \qquad \begin{array}{r} -11 \\ \times\ \ 27 \\ \hline -297 \end{array}$$

$$\begin{array}{r cccccccccccc}
 & & & & & & & 1 & 1 & 0 & 1 & 0 & 1 \\
 & & & & & & \times & +1 & 0 & -1 & +1 & 0 & -1 \\ \cline{8-13}
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
 & & & & & & & & & & & 0 & \\
 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & & \\
\text{sign extension} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & & \\
 & & & & & & & & 0 & & & & \\
 & 1_1 & 1_1 & 1_1 & 0_1 & 1_1 & 0_1 & 1_1 & {}_1 & & & & \\ \cline{2-13}
 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1
\end{array}$$

(*d*)

$$\begin{array}{r} 001111 \\ \times\, 001111 \\ \hline \end{array} \qquad \begin{array}{r} 15 \\ \times\ 15 \\ \hline 225 \end{array}$$

$$\begin{array}{cccccccccccc}
 & & & & & & 0 & 0 & 1 & 1 & 1 & 1 \\
 & & & & & \times & 0 & +1 & 0 & 0 & 0 & -1 \\ \cline{6-12}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & & & & \\ \hline
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1
\end{array}$$

6.18. The multiplication answers are:

(*a*)

$$\begin{array}{r} 010111 \\ \times\, 110110 \\ \hline \end{array}$$

$$\begin{array}{cccccccccccc}
 & & & & & & 0 & 1 & 0 & 1 & 1 & 1 \\
 & & & & & & & -1 & & +2 & & -2 \\ \cline{6-12}
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & & \\
1_1 & 1_1 & 1_1 & 0_2 & 1_1 & 0_1 & 0_1 & 1 & & & & \\ \hline
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0
\end{array}$$

(*b*)

$$\begin{array}{r} 110011 \\ \times\, 101100 \\ \hline \end{array}$$

$$\begin{array}{cccccccccccc}
 & & & & & & 1 & 1 & 0 & 0 & 1 & 1 \\
 & & & & & & & -1 & & -1 & & 0 \\ \cline{6-12}
 & & & & & & & & & & & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & & \\
0 & 0 & 0 & 0_1 & 1_1 & 1_1 & 0_1 & 1 & & & & \\ \hline
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}$$

(*c*)

$$\begin{array}{r} 110101 \\ \times\, 011011 \\ \hline \end{array}$$

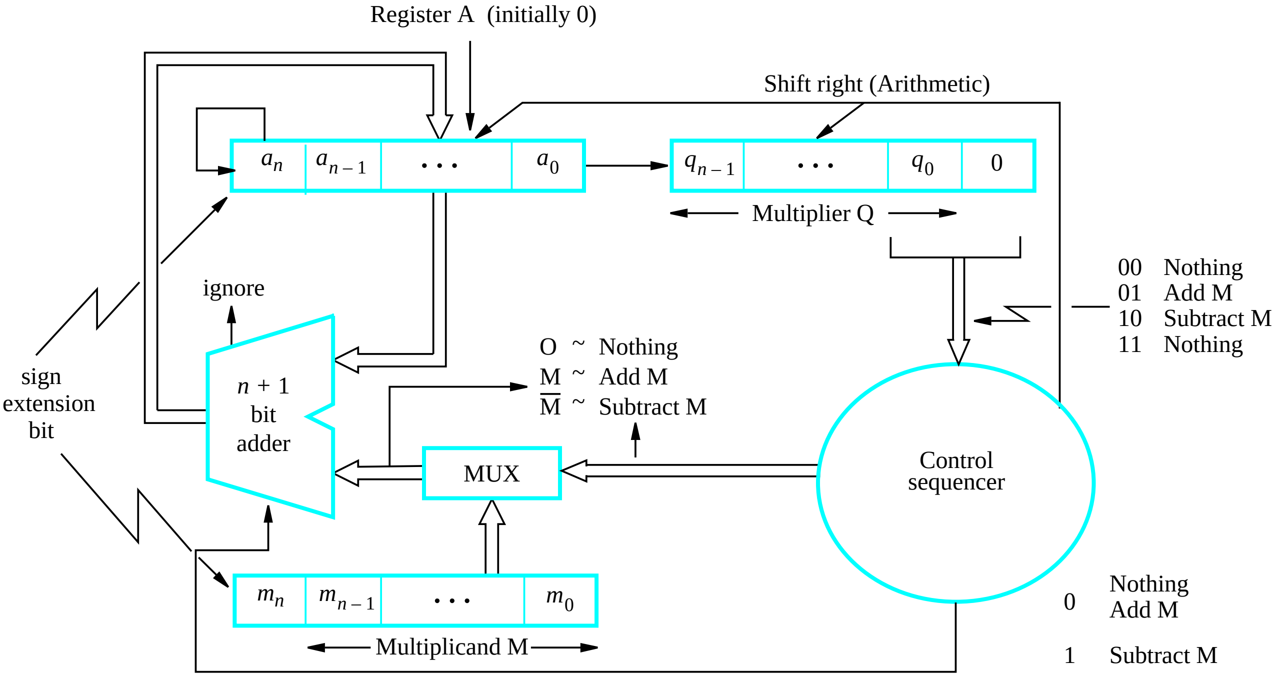
$$\begin{array}{cccccccccccc}
 & & & & & & 1 & 1 & 0 & 1 & 0 & 1 \\
 & & & & & & & +2 & & -1 & & -1 \\ \cline{6-12}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & \\
1 & 1 & 1 & 0 & 1 & 0_1 & 1 & {}_1 & & & & \\ \hline
1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1
\end{array}$$

(*d*)

$$\begin{array}{r} 001111 \\ \times\, 001111 \\ \hline \end{array}$$

$$\begin{array}{cccccccccccc}
 & & & & & & 0 & 0 & 1 & 1 & 1 & 1 \\
 & & & & & & & +1 & & & & -1 \\ \cline{7-12}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & & & & \\ \hline
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1
\end{array}$$

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)

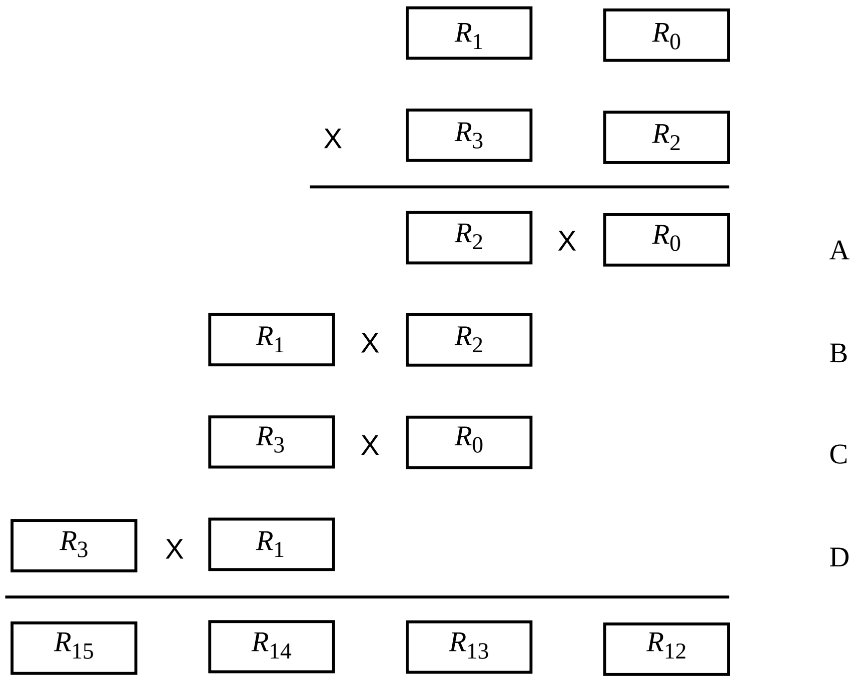
$$\begin{array}{r r}
1110 & -2 \\
\times\ 1101 & \times -3 \\
\hline
1110 & 6 \\
0000 & \\
1000 & \\
0000 & \\
\cline{1-1}
0110 &
\end{array}$$

(b)

$$\begin{array}{r r}
0010 & 2 \\
\times\ 1110 & \times -2 \\
\hline
0000 & -4 \\
0100 & \\
1000 & \\
0000 & \\
\cline{1-1}
1100 &
\end{array}$$

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:

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

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 4×4 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

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

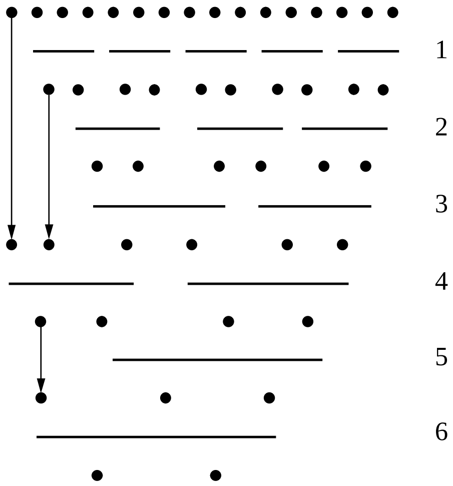
gate delays, including the initial AND gate delay to develop all bit products. The answer is thus 13, as computed directly in Part (a), for the 4×4 case.

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:

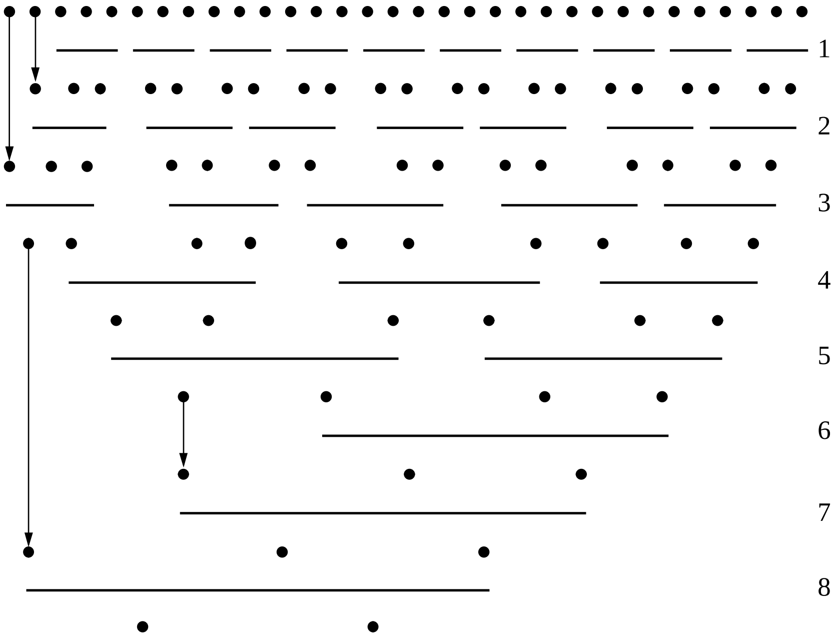
$$\begin{aligned}
\log_2 k + n(\log_2 2 - \log_2 3) &= \log_2 2 \\
\log_2 k &= 1 + n(\log_2 3 - \log_2 2) \\
&= 1 + n(1.59 - 1) \\
n &= \frac{(\log_2 k) - 1}{0.59} \\
&= 1.7\log_2 k - 1.7
\end{aligned}$$

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 ±infinity, 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 ± 1 and $\pm(2^{11} - 1)$; and the smallest and largest fractions (other than 0) are $\pm 2^{-11}$ and approximately ± 1.

(d)

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

“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:

$$\begin{array}{lr} (A) & 1.11111111000 \\ (B) & 0.01001010101 \\ \hline & 10.01001001101 \end{array}$$

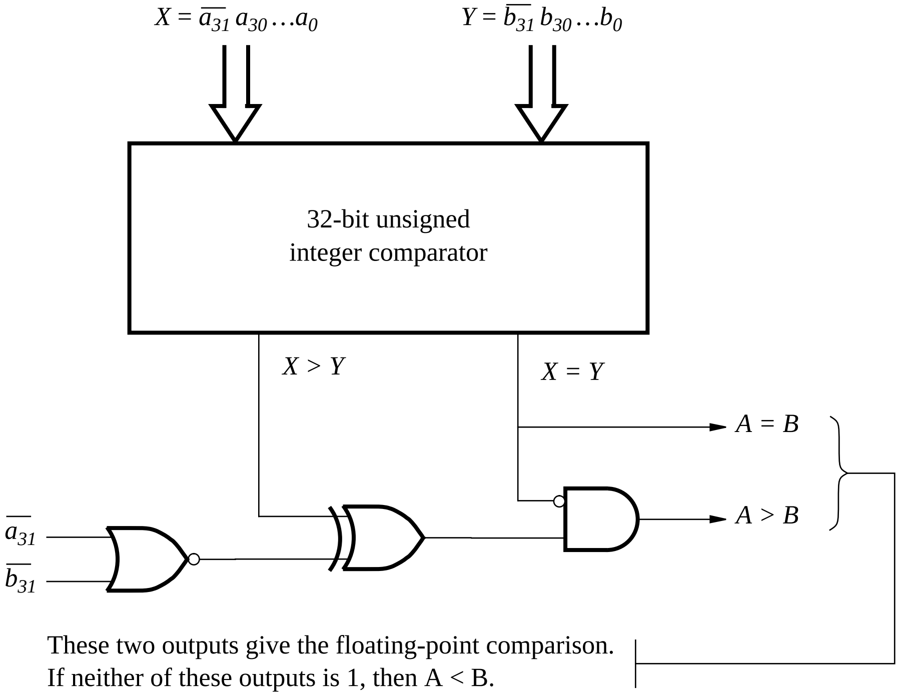
Shift right one position to put in normalized form: 1.001001001101 and increase exponent of sum to 100010. Truncate the mantissa to the right of the binary point to 9 bits by rounding to obtain 001001010. The answer is 0 100010 001001010.

(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 $A = a_{31}a_{30} \ldots a_0$ and $B = b_{31}b_{30} \ldots b_0$ be the two floating-point numbers to be compared.


$X = \overline{a_{31}}\, a_{30} \ldots a_0$
$Y = \overline{b_{31}}\, b_{30} \ldots b_0$
32-bit unsigned integer comparator
$X > Y$
$X = Y$
$A = B$
$A > B$
$\overline{a_{31}}$
$\overline{b_{31}}$

These two outputs give the floating-point comparison.
If neither of these outputs is 1, then A < B.

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

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:

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

With only 2 guard bits, we would have had:

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

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) / SUBTRACT(1) (AS)	S_A	S_B	sign from 8-bit subtractor (8_s)	sign from 25-bit adder/ subtractor (25_s)	ADD/ SUB	S_R
0	0	0	0	0	0	0
				1		d
			1	0		0
				1		d
0	0	1	0	0	1	0
				1		1
			1	0		1
				1		d
0	1	0	0	0	1	1
				1		0
			1	0		0
				1		d
0	1	1	0	0	0	1
				1		d
			1	0		1
				1		d
1	0	0	0	0	1	0
				1		1
			1	0		1
				1		d
1	0	1	0	0	0	0
				1		d
			1	0		0
				1		d
1	1	0	0	0	0	1
				1		d
			1	0		1
				1		d
1	1	1	0	0	1	1
				1		0
			1	0		0
				1		d

these variables determine ADD/SUB

ADD(0)/ SUBTRACT(1) (AS) \ $S_A S_B$	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$\text{ADD/SUB} = AS \oplus S_A \oplus S_B$$

$S_B\ 8_s$ \ $AS\ S_A$	00	01	11	10
00	0	1	1	0
01	0	0	1	1
11	1	1	0	0
10	0	1	1	0

$25_s = 0$

$S_B\ 8_s$ \ $AS\ S_A$	00	01	11	10
00	d	0	d	1
01	d	d	d	d
11	d	d	d	d
10	1	d	0	d

$25_s = 1$

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

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 $n = n_7 n_6 \ldots n_0$, and define an enable signal, EN, as EN = $\overline{n}_7 \overline{n}_6 \overline{n}_5$. This variable must be 1 for any output of the SHIFTER to be non-zero. Let $m = m_{23} m_{22} \ldots m_0$ and $s_{23} s_{22} \ldots s_0$ 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 $i = i_{23} i_{22} \ldots i_0$. We will first shift m into i based on EN and $n_4 n_3$. (Then we will shift i into s, based on $n_2 n_1 n_0$.) Only the part of i needed to generate s_0 will be specified.

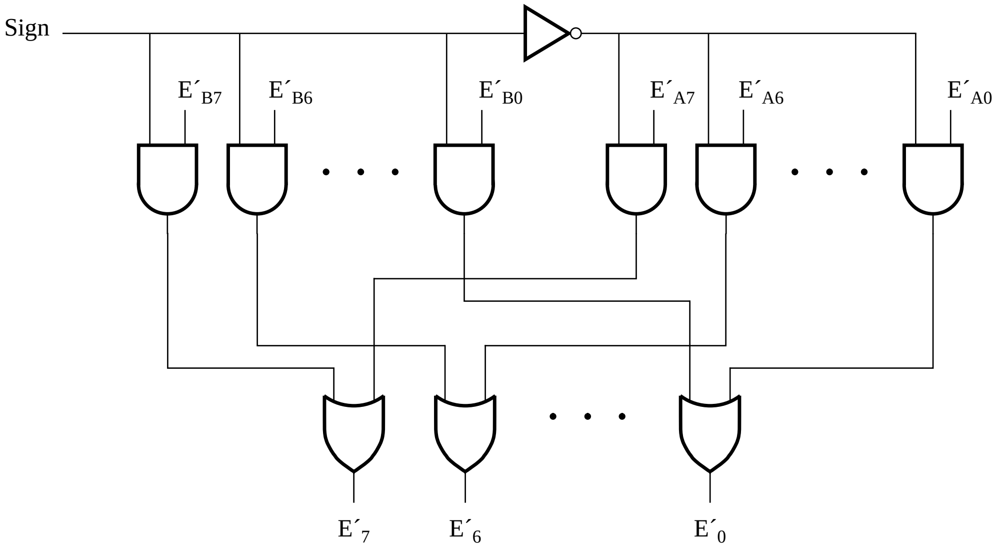
$$\begin{aligned}
i_7 &= \mathrm{EN} n_4 \overline{n}_3 m_{23} + \mathrm{EN} \overline{n}_4 n_3 m_{15} + \mathrm{EN} \overline{n}_4 \overline{n}_3 m_7 \\
i_6 &= (\ldots) m_{22} + (\ldots) m_{14} + (\ldots) m_6 \\
i_5 &= (\ldots) m_{21} + (\ldots) m_{13} + (\ldots) m_5 \\
&\;\cdot \\
&\;\cdot \\
&\;\cdot \\
i_0 &= (\ldots) m_{16} + (\ldots) m_8 + (\ldots) m_0
\end{aligned}$$

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 $n_2 n_1 n_0$ as follows:

$$\begin{aligned}
s_0 = \; & n_2 n_1 n_0 i_7 + n_2 n_1 \overline{n}_0 i_6 + n_2 \overline{n}_1 n_0 i_5 + n_2 \overline{n}_1 \overline{n}_0 i_4 \\
& + \overline{n}_2 n_1 n_0 i_3 + \overline{n}_2 n_1 \overline{n}_0 i_2 + \overline{n}_2 \overline{n}_1 n_0 i_1 + \overline{n}_2 \overline{n}_1 \overline{n}_0 i_0
\end{aligned}$$

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:

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

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 $-1 \le i \le 23$, 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, ..., 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, $\ldots$, 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:

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

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 ns.

7.5. For the organization of Figure 7.1:

(*a*) 1. PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2. Z_{out}, PC_{in}, Y_{in}, WMFC
3. MDR_{out}, IR_{in}
4. PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
5. Z_{out}, PC_{in}, Y_{in}
6. R1_{out}, Y_{in}, WMFC
7. MDR_{out}, SelectY, Add, Z_{in}
8. Z_{out}, R1_{in}, End

(*b*) 1-4. Same as in (*a*)
5. Z_{out}, PC_{in}, WMFC
6. MDR_{out}, MAR_{in}, Read
7. R1_{out}, Y_{in}, WMFC
8. MDR_{out}, Add, Z_{in}
9. Z_{out}, R1_{in}, End

(*c*) 1-5. Same as in (*b*)
6. MDR_{out}, MAR_{in}, Read, 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}, 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

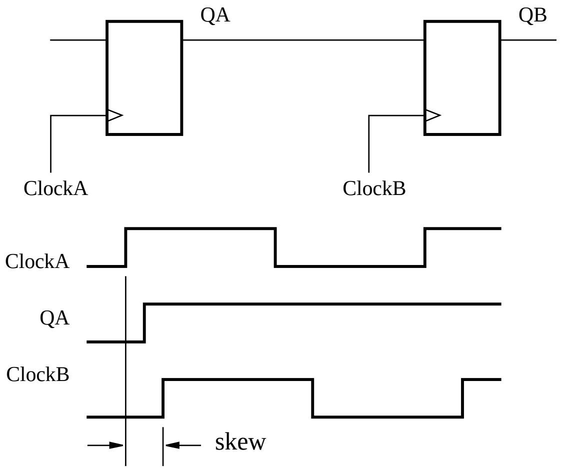
7.11. Let SP be the stack pointer register. The following sequence is for a processor that stores the return address on a stack in the memory.

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

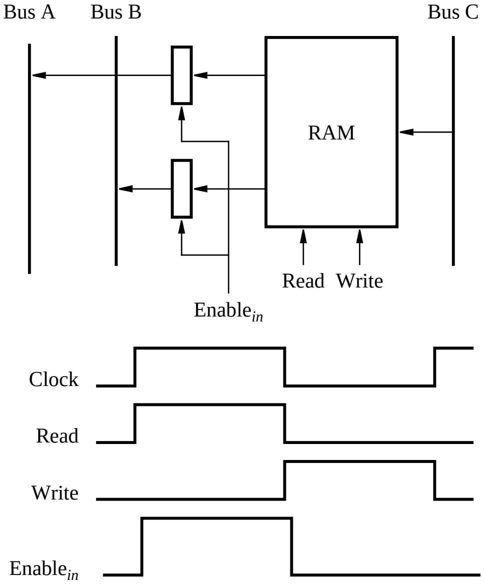
7.13. The latch in Figure A.27 cannot be used to implement a register that can be both the source and the destination of a data transfer operation. For example, it cannot be used to implement register Z in Figure 7.1. It may be used in other registers, provided that hold time requirements are met.

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.

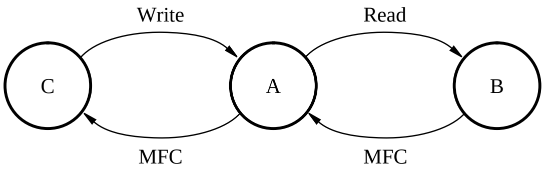


In the absence of clock skew, flip-flop B records a 0 at the first clock edge. However, if Clock B is delayed as shown, the flip-flop records a 1.

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.

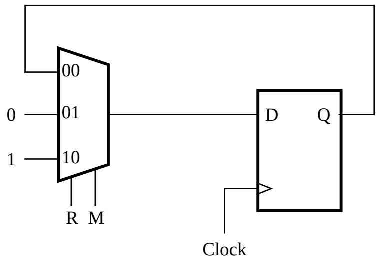


7.16. The step counter advances at the end of a clock period in which Run is equal to 1. With reference to Figure 7.5, Run should be set to 0 during the first clock cycle of step 2 and set to 1 as soon as MFC is received. In general, Run should be set to 0 by WMFC and returned to 1 when MFC is received. 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.

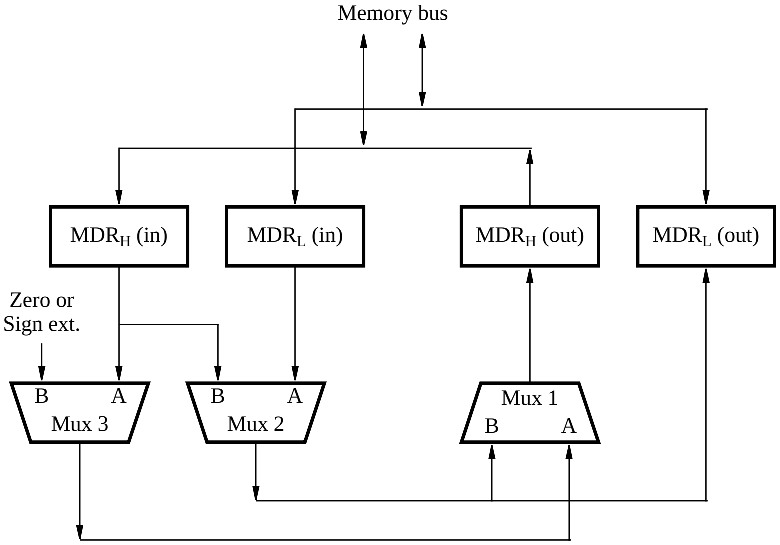


Run = WNFC · (B + C)

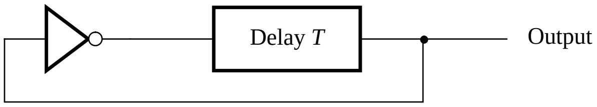
7.17. The following circuit uses a multiplexer arrangement similar to that in Figure 7.3.

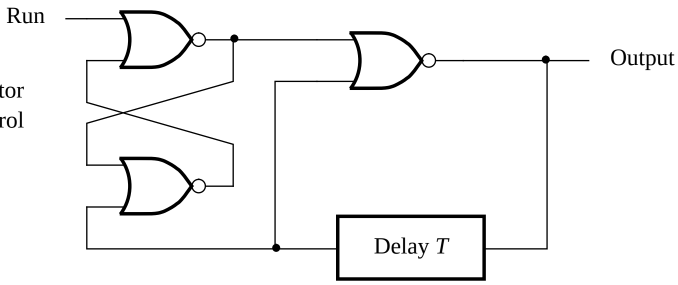


7.18. A possible arrangement is shown below. For clarity, we have assumed that MDR consists of two separate registers for input and output data. Multiplexers Mux-1 and Mux-2 select input B for even and input A for odd byte operations. Mux 3 selects input A for word operations and input B for byte operations. 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.

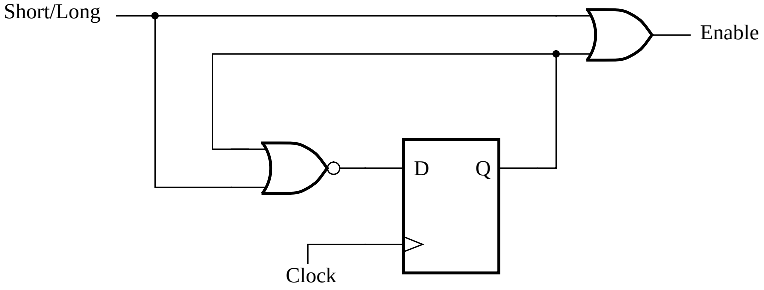


7.19. Use the delay element in a ring oscillator as shown below. The frequency of oscillation is 1/(2T). By adding the control circuit shown, the oscillator will run only while Run is equal to 1. When stopped, its output A is equal to 0. The oscillator will always generate complete output pulses. If 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.

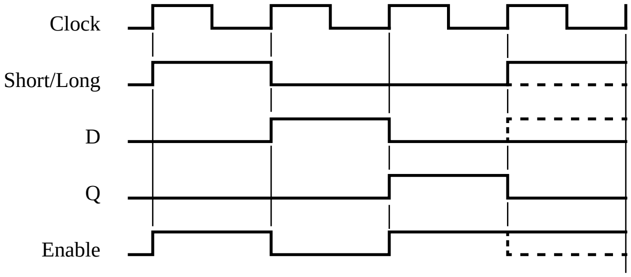

Ring oscillator


Ring oscillator
with run/stop control

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



Short/Long	Q	D
0	0	1
0	1	0
1	0	0
1	1	0



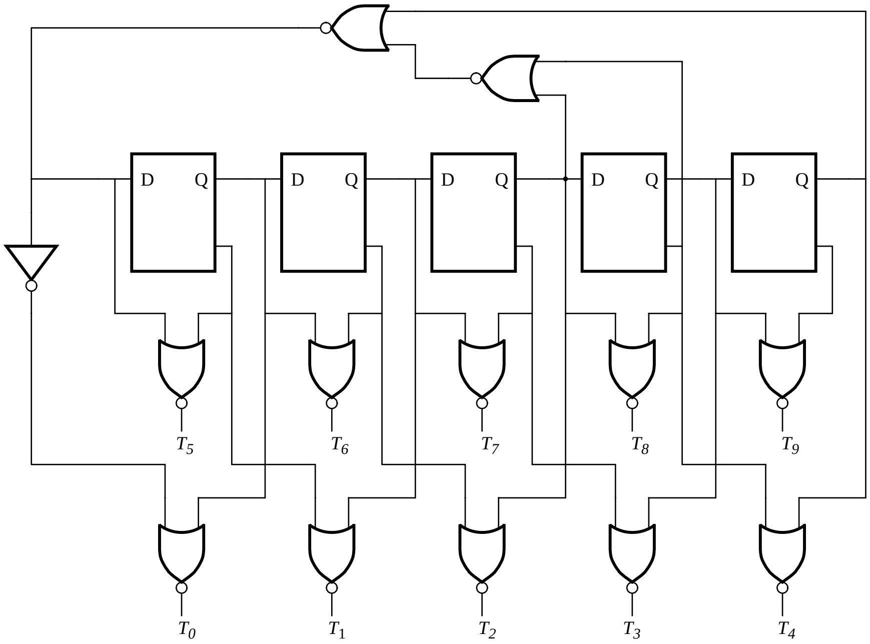
7.21. (*a*) Count sequence is: 0000 1000 1100 1110 1111 0111 0011 0001 0000

(*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

$$D_1 = \overline{Q_5 + \overline{Q_3 + \overline{Q4}}}$$

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.

$$\begin{aligned}
\text{Store} &= (\overline{\mathrm{ARS} + \mathrm{LSR} + \mathrm{SL} + \mathrm{LLD}}) \\
D_{15} &= \mathrm{ASR} \cdot Q_{15} + \mathrm{SL} \cdot Q_{14} + \mathrm{ROR} \cdot \mathrm{Carry} + \mathrm{LD} \cdot D_{15} + \text{Store} \cdot Q_{15} \\
D_1 &= (\mathrm{ASR} + \mathrm{LSR} + \mathrm{ROR}) \cdot Q_2 + \mathrm{SL} \cdot Q_0 + \mathrm{LD} \cdot D_1 + \text{Store} \cdot Q_1 \\
D_0 &= (\mathrm{ASR} + \mathrm{LSR} + \mathrm{ROR}) \cdot Q_1 + \mathrm{LD} \cdot D_0 + + \text{Store} \cdot Q_0
\end{aligned}$$

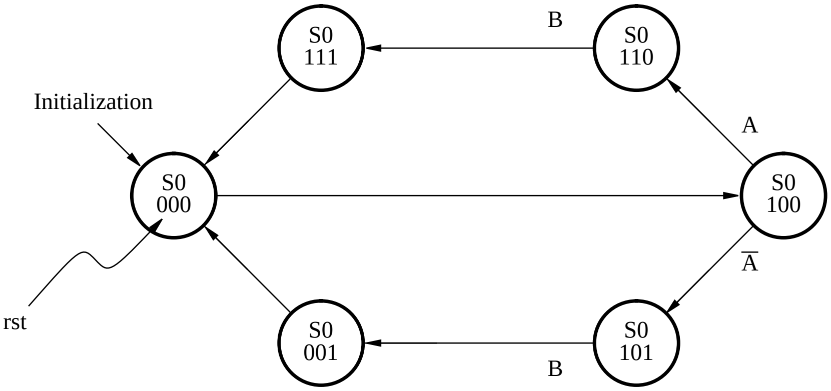
7.23. A state diagram for the required controller is given below. This is a Moore machine. The output values are given inside each state as they are functions of the state only.

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

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

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

$$\begin{aligned} D(r) &= s\,\bar{t}\,B + \bar{s}\,\bar{t} \\ D(s) &= \bar{s}\,\bar{t}\,A + s\,\bar{t}\,B \\ D(t) &= s\,\bar{t}\,B + \bar{s}\,\bar{t}\,\overline{A} + \bar{s}\,t\,B \end{aligned}$$



7.24. Microroutine:

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

7.25. Conditional branch

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

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

Address	**Microinstruction**
(Octal)	
000-002	Same as in Figure 7.21
003	μBranch $\{\mu\mathrm{PC} \leftarrow 300\}$
300	if Z+(N$\oplus$V = 1) then μBranch $\{\mu\mathrm{PC} \leftarrow 000\}$
301	if (N = 1) then μBranch $\{\mu\mathrm{PC} \leftarrow 000\}$
302	PC$_{out}$, Y$_{in}$
303	Offset-field-of-IR$_{out}$, SelectY, Add, Z$_{in}$
304	Z$_{out}$, PC$_{in}$, 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 (Octal)	**Microinstruction**
000-002	Same as in Figure 7.21
003	μBranch $\{\mu\mathrm{PC} \leftarrow 010\}$
010	if (IR$_{10-8}$ $= 000$) then μBranch $\{\mu\mathrm{PC} \leftarrow 101\}$
011	if (IR$_{10-8}$ $= 001$) then μBranch $\{\mu\mathrm{PC} \leftarrow 111\}$
012	if (IR$_{10-9}$ $= 01$) then μBranch $\{\mu\mathrm{PC} \leftarrow 121\}$
013	if (IR$_{10-9}$ $= 10$) then μBranch $\{\mu\mathrm{PC} \leftarrow 141\}$
014	μBranch $\{\mu\mathrm{PC} \leftarrow 161\}$
121	Rsrc$_{out}$, MAR$_{in}$, Read, Select4, Add, Zin
122	Z$_{out}$, Rsrc$_{in}$
123	if (IR$_8$ $= 1$) then μBranch $\{\mu\mathrm{PC} \leftarrow 171\}$
124	μBranch $\{\mu\mathrm{PC} \leftarrow 170\}$
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 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 μBranch 0111
0011	if (b_6b_5) = 01) then μBranch 1010
0100	if (b_6b_5) = 10) then μBranch 1100
0101	I
0110	μBranch 1111
0111	C
1000	D
1001	μBranch 1111
1010	E
1011	μBranch 1111
1100	F
1101	G
1110	H
1111	J

(*b*) Assume that bits b_{6-5} of IR are ORed into bit μPC$_{3-2}$

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

(*c*)

Address	**Microinstruction**	
	Next address	Function
0000	0001	A
0001	0010	B; μPC$_{3-2} \leftarrow b_{6-5}$
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$_{in}\cdot$ 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$_{in}$ $\cdot$ Select4 and Z$_{in}$ $\cdot$ 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$_{out}\cdot$ End, Z$_{out}\cdot$ WMFC. (11 signals)

FB: F2, F3, Instead of Z$_{in}$, MAR$_{in}$, and MDR$_{in}$ use Z$_{in}\cdot$Select4, Z$_{in}\cdot$SelectY, MAR$_{in}$, MAR$_{in}$ $\cdot$ Read, and MDR$_{in}\cdot$ 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

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Pipelining as a means for executing machine instructions concurrently
- Various hazards that cause performance degradation in pipelined processors and means for mitigating their effect
- Hardware and software implications of pipelining
- Influence of pipelining on instruction set design
- Superscalar processors

The basic building blocks of a computer are introduced in preceding chapters. In this chapter, we discuss in detail the concept of pipelining, which is used in modern computers to achieve high performance. We begin by explaining the basics of pipelining and how it can lead to improved performance. Then we examine machine instruction features that facilitate pipelined execution, and we show that the choice of instructions and instruction sequencing can have a significant effect on performance. Pipelined organization requires sophisticated compilation techniques, and *optimizing compilers* have been developed for this purpose. Among other things, such compilers rearrange the sequence of operations to maximize the benefits of pipelined execution.

8.1 BASIC CONCEPTS

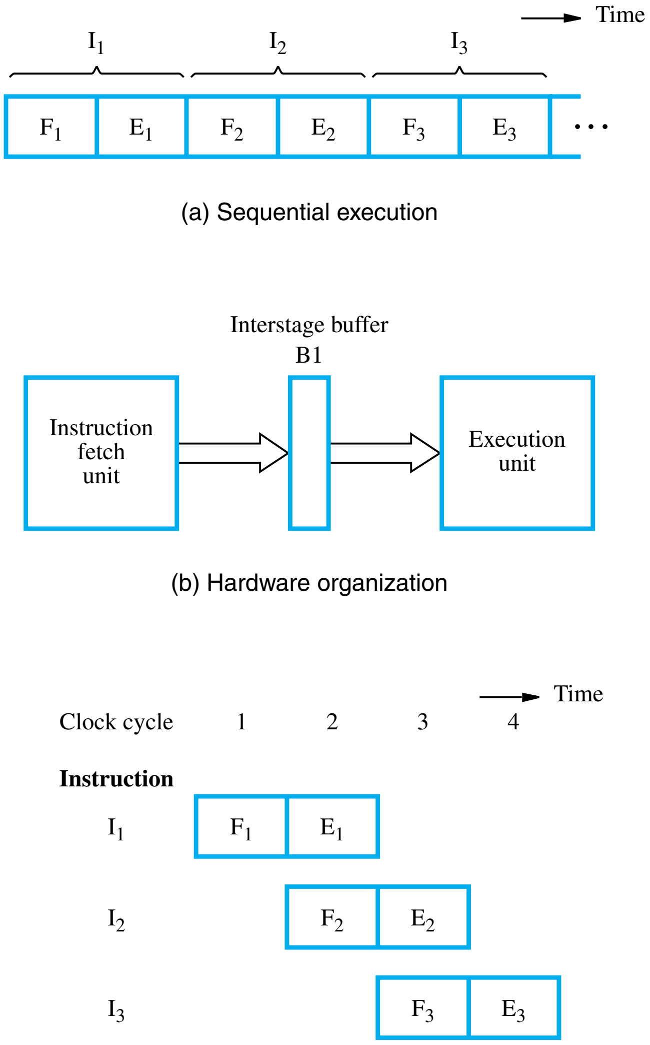
The speed of execution of programs is influenced by many factors. One way to improve performance is to use faster circuit technology to build the processor and the main memory. Another possibility is to arrange the hardware so that more than one operation can be performed at the same time. In this way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

We have encountered concurrent activities several times before. Chapter 1 introduced the concept of multiprogramming and explained how it is possible for I/O transfers and computational activities to proceed simultaneously. DMA devices make this possible because they can perform I/O transfers independently once these transfers are initiated by the processor.

Pipelining is a particularly effective way of organizing concurrent activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly-line operation. Readers are undoubtedly familiar with the assembly line used in car manufacturing. The first station in an assembly line may prepare the chassis of a car, the next station adds the body, the next one installs the engine, and so on. While one group of workers is installing the engine on one car, another group is fitting a car body on the chassis of another car, and yet another group is preparing a new chassis for a third car. It may take days to complete work on a given car, but it is possible to have a new car rolling off the end of the assembly line every few minutes.

Consider how the idea of pipelining can be used in a computer. The processor executes a program by fetching and executing instructions, one after the other. Let F_i and E_i refer to the fetch and execute steps for instruction I_i. Execution of a program consists of a sequence of fetch and execute steps, as shown in Figure 8.1*a*.

Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in Figure 8.1*b*. The instruction fetched by the fetch unit is deposited in an intermediate storage buffer, B1. This buffer is needed to enable the execution unit to execute the instruction while the fetch unit is fetching the next instruction. The results of execution are deposited in the destination location specified by the instruction. For the purposes of this discussion, we assume that both the source and the destination of the data operated on by the instructions are inside the block labeled "Execution unit."



(a) Sequential execution

(b) Hardware organization

(c) Pipelined execution

Figure 8.1 Basic idea of instruction pipelining.

The computer is controlled by a clock whose period is such that the fetch and execute steps of any instruction can each be completed in one clock cycle. Operation of the computer proceeds as in Figure 8.1*c*. In the first clock cycle, the fetch unit fetches an instruction I_1 (step F_1) and stores it in buffer B1 at the end of the clock cycle. In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I_2 (step F_2). Meanwhile, the execution unit performs the operation specified by instruction I_1, which is available to it in buffer B1 (step E_1). By the end of the second clock cycle, the execution of instruction I_1 is completed and instruction I_2 is available. Instruction I_2 is stored in B1, replacing I_1, which is no longer needed. Step E_2 is performed by the execution unit during the third clock cycle, while instruction I_3 is being fetched by the fetch unit. In this manner, both the fetch and execute units are kept busy all the time. If the pattern in Figure 8.1*c* can be sustained for a long time, the completion rate of instruction execution will be twice that achievable by the sequential operation depicted in Figure 8.1*a*.

In summary, the fetch and execute units in Figure 8.1*b* constitute a two-stage pipeline in which each stage performs one step in processing an instruction. An interstage storage buffer, B1, is needed to hold the information being passed from one stage to the next. New information is loaded into this buffer at the end of each clock cycle.

The processing of an instruction need not be divided into only two steps. For example, a pipelined processor may process each instruction in four steps, as follows:

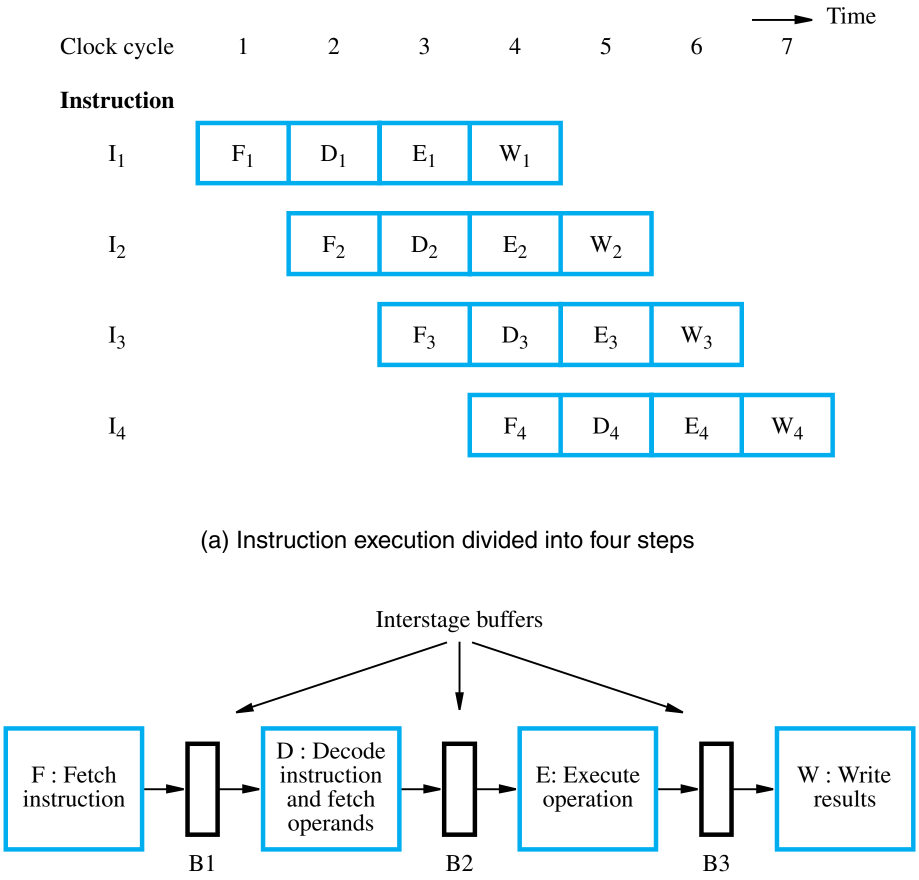
F	Fetch: read the instruction from the memory.
D	Decode: decode the instruction and fetch the source operand(s).
E	Execute: perform the operation specified by the instruction.
W	Write: store the result in the destination location.

The sequence of events for this case is shown in Figure 8.2*a*. Four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Figure 8.2*b*. These units must be capable of performing their tasks simultaneously and without interfering with one another. Information is passed from one unit to the next through a storage buffer. As an instruction progresses through the pipeline, all the information needed by the stages downstream must be passed along. For example, during clock cycle 4, the information in the buffers is as follows:

- Buffer B1 holds instruction I_3, which was fetched in cycle 3 and is being decoded by the instruction-decoding unit.
- Buffer B2 holds both the source operands for instruction I_2 and the specification of the operation to be performed. This is the information produced by the decoding hardware in cycle 3. The buffer also holds the information needed for the write step of instruction I_2 (step W_2). Even though it is not needed by stage E, this information must be passed on to stage W in the following clock cycle to enable that stage to perform the required Write operation.
- Buffer B3 holds the results produced by the execution unit and the destination information for instruction I_1.

8.1.1 Role of Cache Memory

Each stage in a pipeline is expected to complete its operation in one clock cycle. Hence, the clock period should be sufficiently long to complete the task being performed in any stage. If different units require different amounts of time, the clock period must allow the longest task to be completed. A unit that completes its task early is idle for the remainder of the clock period. Hence, pipelining is most effective in improving



(a) Instruction execution divided into four steps

(b) Hardware organization

Figure 8.2 A 4-stage pipeline.

performance if the tasks being performed in different stages require about the same amount of time.

This consideration is particularly important for the instruction fetch step, which is assigned one clock period in Figure 8.2*a*. The clock cycle has to be equal to or greater than the time needed to complete a fetch operation. However, the access time of the main memory may be as much as ten times greater than the time needed to perform basic pipeline stage operations inside the processor, such as adding two numbers. Thus, if each instruction fetch required access to the main memory, pipelining would be of little value.

The use of cache memories solves the memory access problem. In particular, when a cache is included on the same chip as the processor, access time to the cache is usually the same as the time needed to perform other basic operations inside the processor. This makes it possible to divide instruction fetching and processing into steps that are more or less equal in duration. Each of these steps is performed by a different pipeline stage, and the clock period is chosen to correspond to the longest one.

8.1.2 Pipeline Performance

The pipelined processor in Figure 8.2 completes the processing of one instruction in each clock cycle, which means that the rate of instruction processing is four times that of sequential operation. The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages. However, this increase would be achieved only if pipelined operation as depicted in Figure 8.2*a* could be sustained without interruption throughout program execution. Unfortunately, this is not the case.

For a variety of reasons, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time allotted. For example, stage E in the four-stage pipeline of Figure 8.2*b* is responsible for arithmetic and logic operations, and one clock cycle is assigned for this task. Although this may be sufficient for most operations, some operations, such as divide, may require more time to complete. Figure 8.3 shows an example in which the operation specified in instruction I_2 requires three cycles to complete, from cycle 4 through cycle 6. Thus, in cycles 5 and 6, the Write stage must be told to do nothing, because it has no data to work with. Meanwhile, the information in buffer B2 must remain intact until the Execute stage has completed its operation. This means that stage 2 and, in turn, stage 1 are blocked from accepting new instructions because the information in B1 cannot be overwritten. Thus, steps D_4 and F_5 must be postponed as shown.

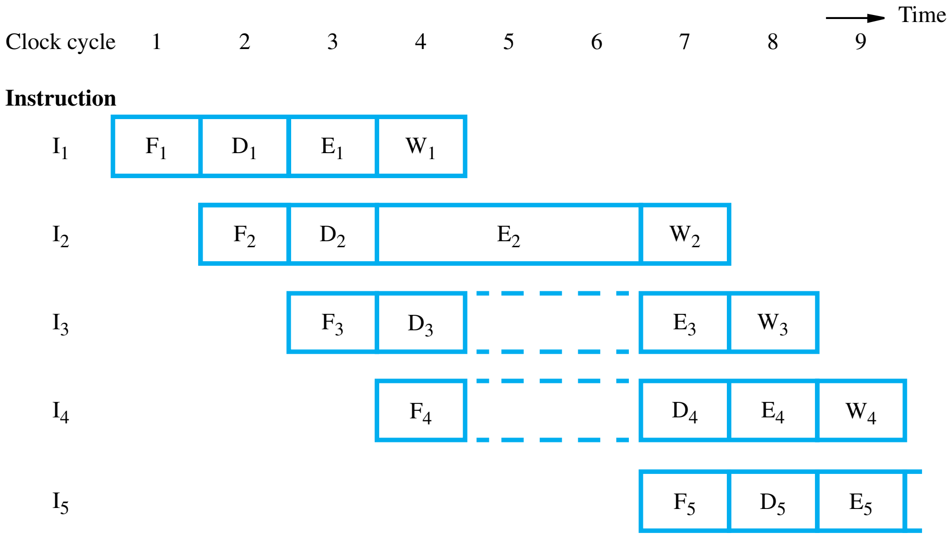
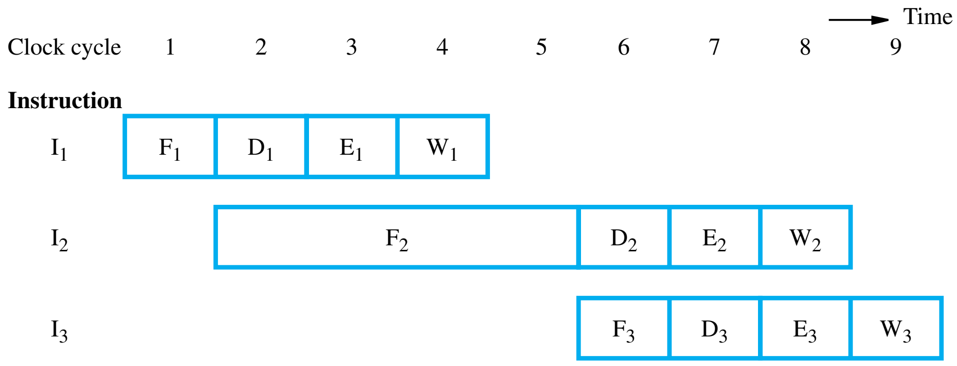


Figure 8.3 Effect of an execution operation taking more than one clock cycle.

Pipelined operation in Figure 8.3 is said to have been *stalled* for two clock cycles. Normal pipelined operation resumes in cycle 7. Any condition that causes the pipeline to stall is called a *hazard*. We have just seen an example of a *data hazard*. A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result some operation has to be delayed, and the pipeline stalls.

The pipeline may also be stalled because of a delay in the availability of an instruction. For example, this may be a result of a miss in the cache, requiring the instruction to be fetched from the main memory. Such hazards are often called *control hazards* or *instruction hazards*. The effect of a cache miss on pipelined operation is illustrated in Figure 8.4. Instruction I_1 is fetched from the cache in cycle 1, and its execution proceeds normally. However, the fetch operation for instruction I_2, which is started in cycle 2, results in a cache miss. The instruction fetch unit must now suspend any further fetch requests and wait for I_2 to arrive. We assume that instruction I_2 is received and loaded into buffer B1 at the end of cycle 5. The pipeline resumes its normal operation at that point.



(a) Instruction execution steps in successive clock cycles

→ Time

Clock cycle	1	2	3	4	5	6	7	8	9
Stage									
F: Fetch	F_1	F_2	F_2	F_2	F_2	F_3			
D: Decode		D_1	idle	idle	idle	D_2	D_3		
E: Execute			E_1	idle	idle	idle	E_2	E_3	
W: Write				W_1	idle	idle	idle	W_2	W_3

(b) Function performed by each processor stage in successive clock cycles

Figure 8.4 Pipeline stall caused by a cache miss in F2.

An alternative representation of the operation of a pipeline in the case of a cache miss is shown in Figure 8.4*b*. This figure gives the function performed by each pipeline stage in each clock cycle. Note that the Decode unit is idle in cycles 3 through 5, the Execute unit is idle in cycles 4 through 6, and the Write unit is idle in cycles 5 through 7. Such idle periods are called *stalls*. They are also often referred to as *bubbles* in the pipeline. Once created as a result of a delay in one of the pipeline stages, a bubble moves downstream until it reaches the last unit.

A third type of hazard that may be encountered in pipelined operation is known as a *structural hazard*. This is the situation when two instructions require the use of a given hardware resource at the same time. The most common case in which this hazard may arise is in access to memory. One instruction may need to access memory as part of the Execute or Write stage while another instruction is being fetched. If instructions and data reside in the same cache unit, only one instruction can proceed and the other instruction is delayed. Many processors use separate instruction and data caches to avoid this delay.

An example of a structural hazard is shown in Figure 8.5. This figure shows how the load instruction

Load X(R1),R2

can be accommodated in our example 4-stage pipeline. The memory address, X+[R1], is computed in step E_2 in cycle 4, then memory access takes place in cycle 5. The operand read from memory is written into register R2 in cycle 6. This means that the execution step of this instruction takes two clock cycles (cycles 4 and 5). It causes the pipeline to stall for one cycle, because both instructions I_2 and I_3 require access to the register file in cycle 6. Even though the instructions and their data are all available, the pipeline is

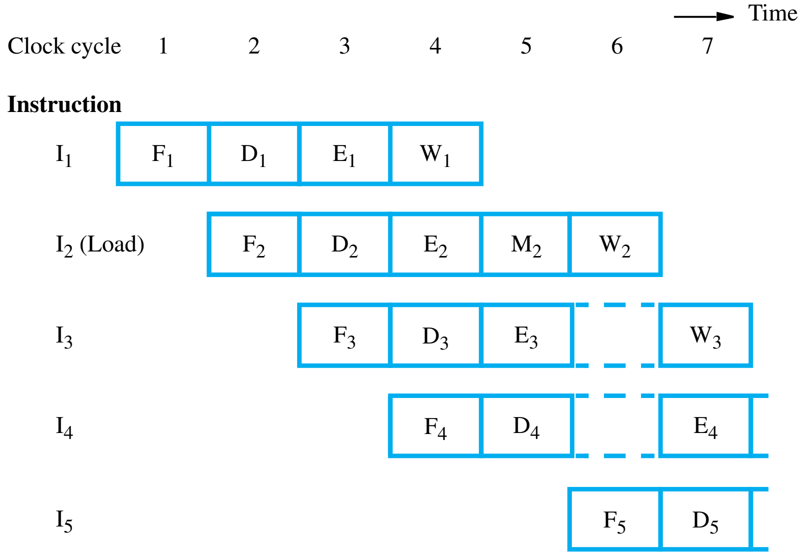


Figure 8.5 Effect of a Load instruction on pipeline timing.

stalled because one hardware resource, the register file, cannot handle two operations at once. If the register file had two input ports, that is, if it allowed two simultaneous write operations, the pipeline would not be stalled. In general, structural hazards are avoided by providing sufficient hardware resources on the processor chip.

It is important to understand that pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases, where throughput is measured by the rate at which instruction execution is completed. Any time one of the stages in the pipeline cannot complete its operation in one clock cycle, the pipeline stalls, and some degradation in performance occurs. Thus, the performance level of one instruction completion in each clock cycle is actually the upper limit for the throughput achievable in a pipelined processor organized as in Figure 8.2*b*.

An important goal in designing processors is to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact. In the following sections we discuss various hazards, starting with data hazards, followed by control hazards. In each case we present some of the techniques used to mitigate their negative effect on performance. We return to the issue of performance assessment in Section 8.8.

8.2 DATA HAZARDS

A data hazard is a situation in which the pipeline is stalled because the data to be operated on are delayed for some reason, as illustrated in Figure 8.3. We will now examine the issue of availability of data in some detail.

Consider a program that contains two instructions, I_1 followed by I_2. When this program is executed in a pipeline, the execution of I_2 can begin before the execution of I_1 is completed. This means that the results generated by I_1 may not be available for use by I_2. We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially. The potential for obtaining incorrect results when operations are performed concurrently can be demonstrated by a simple example. Assume that $A = 5$, and consider the following two operations:

$$A \leftarrow 3 + A$$

$$B \leftarrow 4 \times A$$

When these operations are performed in the order given, the result is $B = 32$. But if they are performed concurrently, the value of A used in computing B would be the original value, 5, leading to an incorrect result. If these two operations are performed by instructions in a program, then the instructions must be executed one after the other, because the data used in the second instruction depend on the result of the first instruction. On the other hand, the two operations

$$A \leftarrow 5 \times C$$

$$B \leftarrow 20 + C$$

can be performed concurrently, because these operations are independent.

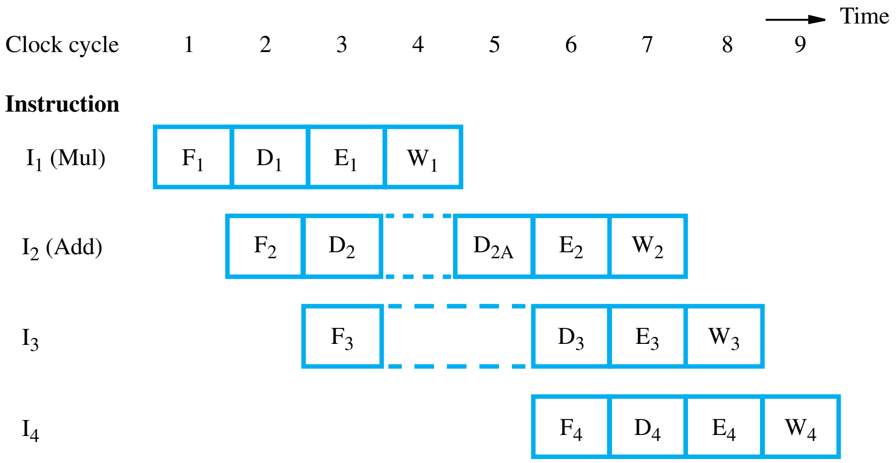


Figure 8.6 Pipeline stalled by data dependency between D_2 and W_1.

This example illustrates a basic constraint that must be enforced to guarantee correct results. When two operations depend on each other, they must be performed sequentially in the correct order. This rather obvious condition has far-reaching consequences. Understanding its implications is the key to understanding the variety of design alternatives and trade-offs encountered in pipelined computers.

Consider the pipeline in Figure 8.2. The data dependency just described arises when the destination of one instruction is used as a source in the next instruction. For example, the two instructions

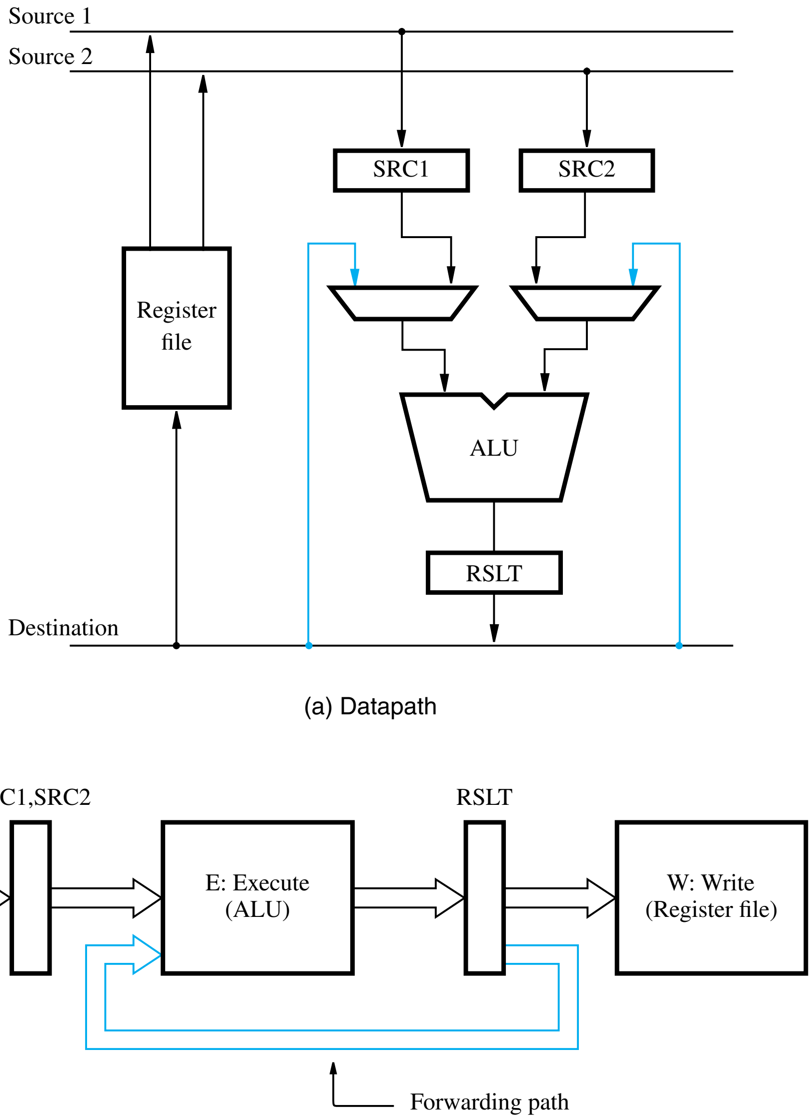
```
Mul   R2,R3,R4
Add   R5,R4,R6
```

give rise to a data dependency. The result of the multiply instruction is placed into register R4, which in turn is one of the two source operands of the Add instruction. Assuming that the multiply operation takes one clock cycle to complete, execution would proceed as shown in Figure 8.6. As the Decode unit decodes the Add instruction in cycle 3, it realizes that R4 is used as a source operand. Hence, the D step of that instruction cannot be completed until the W step of the multiply instruction has been completed. Completion of step D_2 must be delayed to clock cycle 5, and is shown as step D_{2A} in the figure. Instruction I_3 is fetched in cycle 3, but its decoding must be delayed because step D_3 cannot precede D_2. Hence, pipelined execution is stalled for two cycles.

8.2.1 Operand Forwarding

The data hazard just described arises because one instruction, instruction I_2 in Figure 8.6, is waiting for data to be written in the register file. However, these data are available at the output of the ALU once the Execute stage completes step E_1. Hence, the delay can be reduced, or possibly eliminated, if we arrange for the result of instruction I_1 to be forwarded directly for use in step E_2.

Figure 8.7*a* shows a part of the processor datapath involving the ALU and the register file. This arrangement is similar to the three-bus structure in Figure 7.8, except that registers SRC1, SRC2, and RSLT have been added. These registers constitute the



(a) Datapath

(b) Position of the source and result registers in the processor pipeline

Figure 8.7 Operand forwarding in a pipelined processor.

interstage buffers needed for pipelined operation, as illustrated in Figure 8.7*b*. With reference to Figure 8.2*b*, registers SRC1 and SRC2 are part of buffer B2 and RSLT is part of B3. The data forwarding mechanism is provided by the blue connection lines. The two multiplexers connected at the inputs to the ALU allow the data on the destination bus to be selected instead of the contents of either the SRC1 or SRC2 register.

When the instructions in Figure 8.6 are executed in the datapath of Figure 8.7, the operations performed in each clock cycle are as follows. After decoding instruction I_2 and detecting the data dependency, a decision is made to use data forwarding. The operand not involved in the dependency, register R2, is read and loaded in register SRC1 in clock cycle 3. In the next clock cycle, the product produced by instruction I_1 is available in register RSLT, and because of the forwarding connection, it can be used in step E_2. Hence, execution of I_2 proceeds without interruption.

8.2.2 Handling Data Hazards in Software

In Figure 8.6, we assumed the data dependency is discovered by the hardware while the instruction is being decoded. The control hardware delays reading register R4 until cycle 5, thus introducing a 2-cycle stall unless operand forwarding is used. An alternative approach is to leave the task of detecting data dependencies and dealing with them to the software. In this case, the compiler can introduce the two-cycle delay needed between instructions I_1 and I_2 by inserting NOP (No-operation) instructions, as follows:

```
I1:  Mul   R2,R3,R4
     NOP
     NOP
I2:  Add   R5,R4,R6
```

If the responsibility for detecting such dependencies is left entirely to the software, the compiler must insert the NOP instructions to obtain a correct result. This possibility illustrates the close link between the compiler and the hardware. A particular feature can be either implemented in hardware or left to the compiler. Leaving tasks such as inserting NOP instructions to the compiler leads to simpler hardware. Being aware of the need for a delay, the compiler can attempt to reorder instructions to perform useful tasks in the NOP slots, and thus achieve better performance. On the other hand, the insertion of NOP instructions leads to larger code size. Also, it is often the case that a given processor architecture has several hardware implementations, offering different features. NOP instructions inserted to satisfy the requirements of one implementation may not be needed and, hence, would lead to reduced performance on a different implementation.

8.2.3 Side Effects

The data dependencies encountered in the preceding examples are explicit and easily detected because the register involved is named as the destination in instruction I_1 and as a source in I_2. Sometimes an instruction changes the contents of a register other than the one named as the destination. An instruction that uses an autoincrement or autodecrement addressing mode is an example. In addition to storing new data in its destination location, the instruction changes the contents of a source register used to access one of its operands. All the precautions needed to handle data dependencies involving the destination location must also be applied to the registers affected by an autoincrement or autodecrement operation. When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a *side effect.* For example, stack instructions, such as push and pop, produce similar side effects because they implicitly use the autoincrement and autodecrement addressing modes.

Another possible side effect involves the condition code flags, which are used by instructions such as conditional branches and add-with-carry. Suppose that registers R1 and R2 hold a double-precision integer number that we wish to add to another double-precision number in registers R3 and R4. This may be accomplished as follows:

```
Add             R1,R3
AddWithCarry    R2,R4
```

An implicit dependency exists between these two instructions through the carry flag. This flag is set by the first instruction and used in the second instruction, which performs the operation

$$R4 \leftarrow [R2] + [R4] + \text{carry}$$

Instructions that have side effects give rise to multiple data dependencies, which lead to a substantial increase in the complexity of the hardware or software needed to resolve them. For this reason, instructions designed for execution on pipelined hardware should have few side effects. Ideally, only the contents of the destination location, either a register or a memory location, should be affected by any given instruction. Side effects, such as setting the condition code flags or updating the contents of an address pointer, should be kept to a minimum. However, Chapter 2 showed that the autoincrement and autodecrement addressing modes are potentially useful. Condition code flags are also needed for recording such information as the generation of a carry or the occurrence of overflow in an arithmetic operation. In Section 8.4 we show how such functions can be provided by other means that are consistent with a pipelined organization and with the requirements of optimizing compilers.

8.3 INSTRUCTION HAZARDS

The purpose of the instruction fetch unit is to supply the execution units with a steady stream of instructions. Whenever this stream is interrupted, the pipeline stalls, as Figure 8.4 illustrates for the case of a cache miss. A branch instruction may also cause the pipeline to stall. We will now examine the effect of branch instructions and the techniques that can be used for mitigating their impact. We start with unconditional branches.

8.3.1 UNCONDITIONAL BRANCHES

Figure 8.8 shows a sequence of instructions being executed in a two-stage pipeline. Instructions I_1 to I_3 are stored at successive memory addresses, and I_2 is a branch instruction. Let the branch target be instruction I_k. In clock cycle 3, the fetch operation for instruction I_3 is in progress at the same time that the branch instruction is being decoded and the target address computed. In clock cycle 4, the processor must discard I_3, which has been incorrectly fetched, and fetch instruction I_k. In the meantime, the hardware unit responsible for the Execute (E) step must be told to do nothing during that clock period. Thus, the pipeline is stalled for one clock cycle.

The time lost as a result of a branch instruction is often referred to as the *branch penalty*. In Figure 8.8, the branch penalty is one clock cycle. For a longer pipeline, the branch penalty may be higher. For example, Figure 8.9*a* shows the effect of a branch instruction on a four-stage pipeline. We have assumed that the branch address is computed in step E_2. Instructions I_3 and I_4 must be discarded, and the target instruction, I_k, is fetched in clock cycle 5. Thus, the branch penalty is two clock cycles.

Reducing the branch penalty requires the branch address to be computed earlier in the pipeline. Typically, the instruction fetch unit has dedicated hardware to identify a branch instruction and compute the branch target address as quickly as possible after an instruction is fetched. With this additional hardware, both of these tasks can be performed in step D_2, leading to the sequence of events shown in Figure 8.9*b*. In this case, the branch penalty is only one clock cycle.

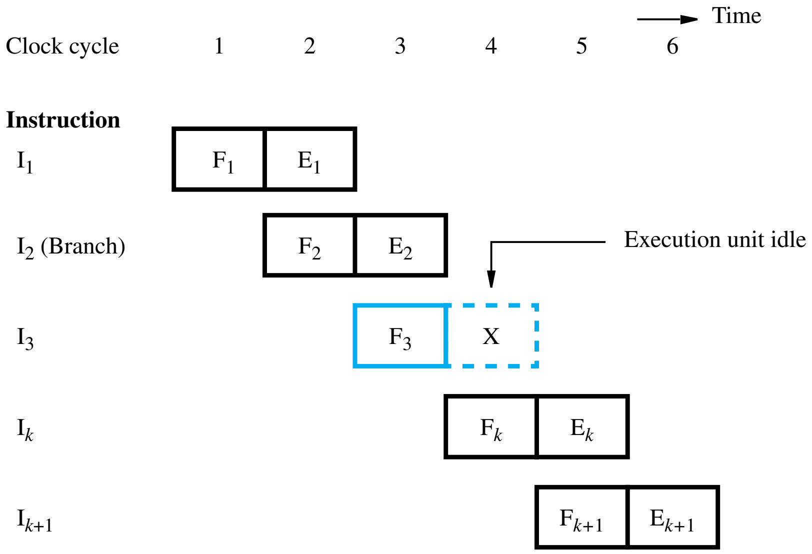
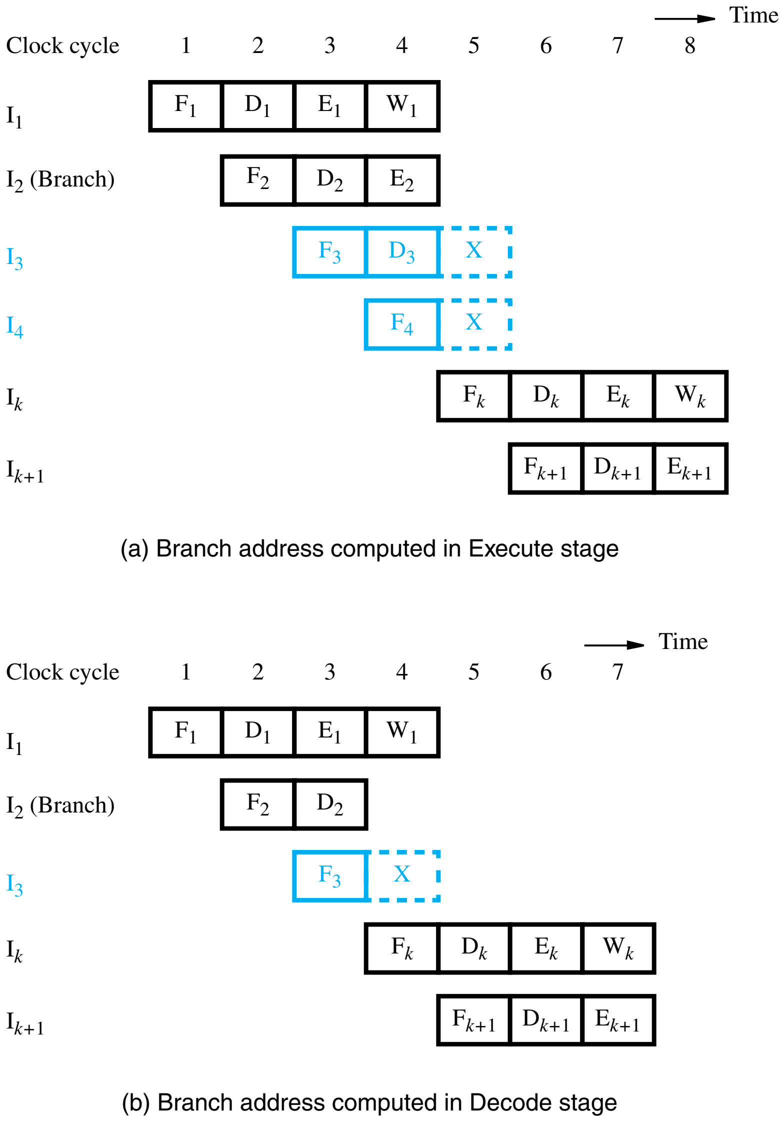


Figure 8.8 An idle cycle caused by a branch instruction.



(a) Branch address computed in Execute stage

(b) Branch address computed in Decode stage

Figure 8.9 Branch timing.

Instruction Queue and Prefetching

Either a cache miss or a branch instruction stalls the pipeline for one or more clock cycles. To reduce the effect of these interruptions, many processors employ sophisticated fetch units that can fetch instructions before they are needed and put them in a queue. Typically, the instruction queue can store several instructions. A separate unit, which we call the *dispatch unit,* takes instructions from the front of the queue and

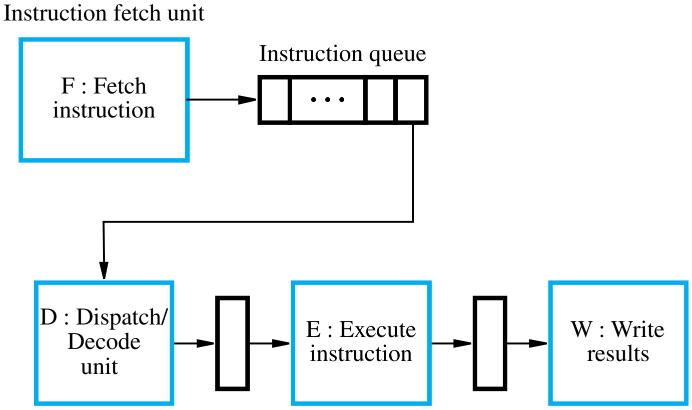


Figure 8.10 Use of an instruction queue in the hardware organization of Figure 8.2*b*.

sends them to the execution unit. This leads to the organization shown in Figure 8.10. The dispatch unit also performs the decoding function.

To be effective, the fetch unit must have sufficient decoding and processing capability to recognize and execute branch instructions. It attempts to keep the instruction queue filled at all times to reduce the impact of occasional delays when fetching instructions. When the pipeline stalls because of a data hazard, for example, the dispatch unit is not able to issue instructions from the instruction queue. However, the fetch unit continues to fetch instructions and add them to the queue. Conversely, if there is a delay in fetching instructions because of a branch or a cache miss, the dispatch unit continues to issue instructions from the instruction queue.

Figure 8.11 illustrates how the queue length changes and how it affects the relationship between different pipeline stages. We have assumed that initially the queue contains one instruction. Every fetch operation adds one instruction to the queue and every dispatch operation reduces the queue length by one. Hence, the queue length remains the same for the first four clock cycles. (There is both an F and a D step in each of these cycles.) Suppose that instruction I_1 introduces a 2-cycle stall. Since space is available in the queue, the fetch unit continues to fetch instructions and the queue length rises to 3 in clock cycle 6.

Instruction I_5 is a branch instruction. Its target instruction, I_k, is fetched in cycle 7, and instruction I_6 is discarded. The branch instruction would normally cause a stall in cycle 7 as a result of discarding instruction I_6. Instead, instruction I_4 is dispatched from the queue to the decoding stage. After discarding I_6, the queue length drops to 1 in cycle 8. The queue length will be at this value until another stall is encountered.

Now observe the sequence of instruction completions in Figure 8.11. Instructions I_1, I_2, I_3, I_4, and I_k complete execution in successive clock cycles. Hence, the branch instruction does not increase the overall execution time. This is because the instruction fetch unit has executed the branch instruction (by computing the branch address) concurrently with the execution of other instructions. This technique is referred to as *branch folding*.

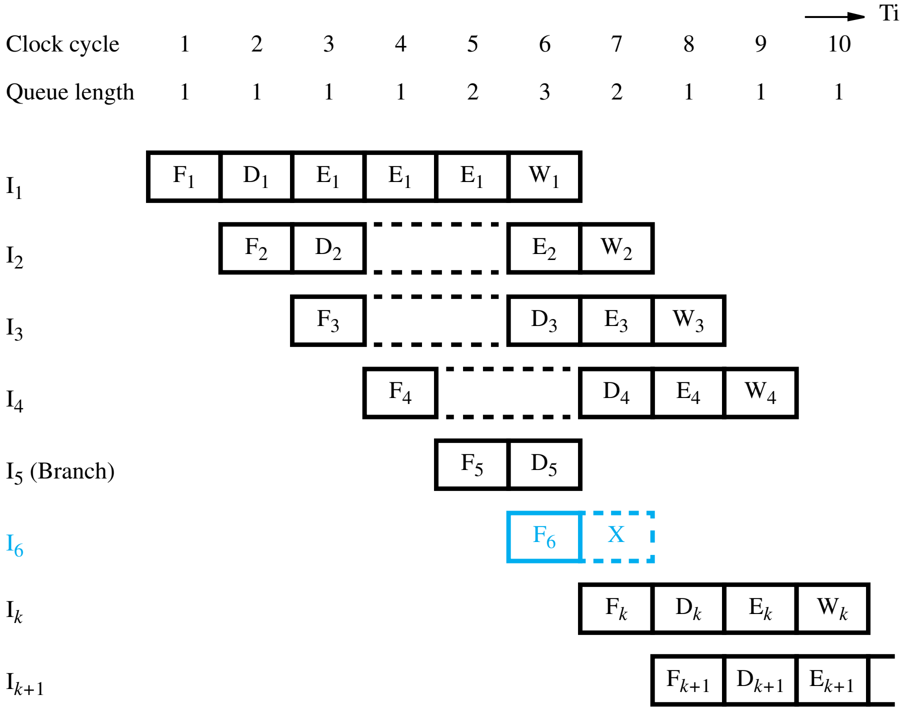


Figure 8.11 Branch timing in the presence of an instruction queue. Branch target address is computed in the D stage.

Note that branch folding occurs only if at the time a branch instruction is encountered, at least one instruction is available in the queue other than the branch instruction. If only the branch instruction is in the queue, execution would proceed as in Figure 8.9*b*. Therefore, it is desirable to arrange for the queue to be full most of the time, to ensure an adequate supply of instructions for processing. This can be achieved by increasing the rate at which the fetch unit reads instructions from the cache. In many processors, the width of the connection between the fetch unit and the instruction cache allows reading more than one instruction in each clock cycle. If the fetch unit replenishes the instruction queue quickly after a branch has occurred, the probability that branch folding will occur increases.

Having an instruction queue is also beneficial in dealing with cache misses. When a cache miss occurs, the dispatch unit continues to send instructions for execution as long as the instruction queue is not empty. Meanwhile, the desired cache block is read from the main memory or from a secondary cache. When fetch operations are resumed, the instruction queue is refilled. If the queue does not become empty, a cache miss will have no effect on the rate of instruction execution.

In summary, the instruction queue mitigates the impact of branch instructions on performance through the process of branch folding. It has a similar effect on stalls caused by cache misses. The effectiveness of this technique is enhanced when the instruction fetch unit is able to read more than one instruction at a time from the instruction cache.

8.3.2 Conditional Branches and Branch Prediction

A conditional branch instruction introduces the added hazard caused by the dependency of the branch condition on the result of a preceding instruction. The decision to branch cannot be made until the execution of that instruction has been completed.

Branch instructions occur frequently. In fact, they represent about 20 percent of the dynamic instruction count of most programs. (The dynamic count is the number of instruction executions, taking into account the fact that some program instructions are executed many times because of loops.) Because of the branch penalty, this large percentage would reduce the gain in performance expected from pipelining. Fortunately, branch instructions can be handled in several ways to reduce their negative impact on the rate of execution of instructions.

Delayed Branch

In Figure 8.8, the processor fetches instruction I_3 before it determines whether the current instruction, I_2, is a branch instruction. When execution of I_2 is completed and a branch is to be made, the processor must discard I_3 and fetch the instruction at the branch target. The location following a branch instruction is called a *branch delay slot*. There may be more than one branch delay slot, depending on the time it takes to execute a branch instruction. For example, there are two branch delay slots in Figure 8.9*a* and one delay slot in Figure 8.9*b*. The instructions in the delay slots are always fetched and at least partially executed before the branch decision is made and the branch target address is computed.

A technique called *delayed branching* can minimize the penalty incurred as a result of conditional branch instructions. The idea is simple. The instructions in the delay slots are always fetched. Therefore, we would like to arrange for them to be fully executed whether or not the branch is taken. The objective is to be able to place useful instructions in these slots. If no useful instructions can be placed in the delay slots, these slots must be filled with NOP instructions. This situation is exactly the same as in the case of data dependency discussed in Section 8.2.

Consider the instruction sequence given in Figure 8.12*a*. Register R2 is used as a counter to determine the number of times the contents of register R1 are shifted left. For a processor with one delay slot, the instructions can be reordered as shown in Figure 8.12*b*. The shift instruction is fetched while the branch instruction is being executed. After evaluating the branch condition, the processor fetches the instruction at LOOP or at NEXT, depending on whether the branch condition is true or false, respectively. In either case, it completes execution of the shift instruction. The sequence of events during the last two passes in the loop is illustrated in Figure 8.13. Pipelined operation is not interrupted at any time, and there are no idle cycles. Logically, the program is executed as if the branch instruction were placed after the shift instruction. That is, branching takes place one instruction later than where the branch instruction appears in the instruction sequence in the memory, hence the name “delayed branch.”

```
LOOP    Shift_left    R1
        Decrement     R2
        Branch=0      LOOP
NEXT    Add           R1,R3
```

(a) Original program loop

```
LOOP    Decrement     R2
        Branch=0      LOOP
        Shift_left    R1
NEXT    Add           R1,R3
```

(b) Reordered instructions

Figure 8.12 Reordering of instructions for a delayed branch.

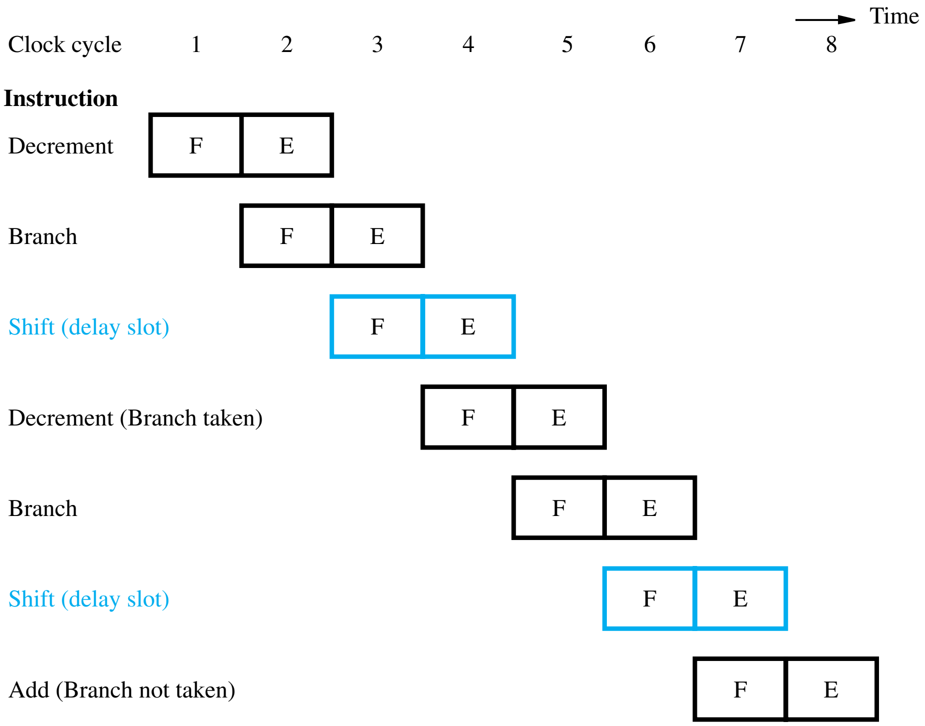


Figure 8.13 Execution timing showing the delay slot being filled during the last two passes through the loop in Figure 8.12*b*.

The effectiveness of the delayed branch approach depends on how often it is possible to reorder instructions as in Figure 8.12. Experimental data collected from many programs indicate that sophisticated compilation techniques can use one branch delay slot in as many as 85 percent of the cases. For a processor with two branch delay slots, the compiler attempts to find two instructions preceding the branch instruction that it can move into the delay slots without introducing a logical error. The chances of finding two such instructions are considerably less than the chances of finding one. Thus, if increasing the number of pipeline stages involves an increase in the number of branch delay slots, the potential gain in performance may not be fully realized.

Branch Prediction

Another technique for reducing the branch penalty associated with conditional branches is to attempt to predict whether or not a particular branch will be taken. The simplest form of branch prediction is to assume that the branch will not take place and to continue to fetch instructions in sequential address order. Until the branch condition is evaluated, instruction execution along the predicted path must be done on a speculative basis. *Speculative execution* means that instructions are executed before the processor is certain that they are in the correct execution sequence. Hence, care must be taken that no processor registers or memory locations are updated until it is confirmed that these instructions should indeed be executed. If the branch decision indicates otherwise, the instructions and all their associated data in the execution units must be purged, and the correct instructions fetched and executed.

An incorrectly predicted branch is illustrated in Figure 8.14 for a four-stage pipeline. The figure shows a Compare instruction followed by a Branch>0 instruction. Branch

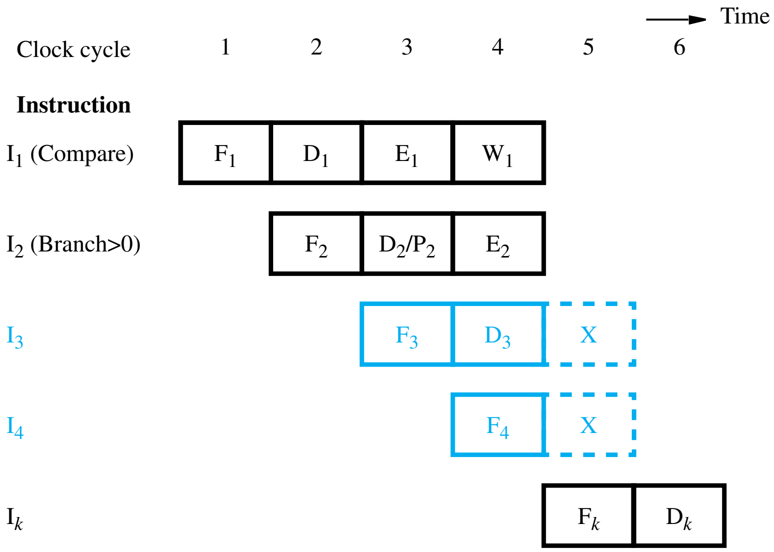


Figure 8.14 Timing when a branch decision has been incorrectly predicted as not taken.

prediction takes place in cycle 3, while instruction I_3 is being fetched. The fetch unit predicts that the branch will not be taken, and it continues to fetch instruction I_4 as I_3 enters the Decode stage. The results of the compare operation are available at the end of cycle 3. Assuming that they are forwarded immediately to the instruction fetch unit, the branch condition is evaluated in cycle 4. At this point, the instruction fetch unit realizes that the prediction was incorrect, and the two instructions in the execution pipe are purged. A new instruction, I_k, is fetched from the branch target address in clock cycle 5.

If branch outcomes were random, then half the branches would be taken. Then the simple approach of assuming that branches will not be taken would save the time lost to conditional branches 50 percent of the time. However, better performance can be achieved if we arrange for some branch instructions to be predicted as taken and others as not taken, depending on the expected program behavior. For example, a branch instruction at the end of a loop causes a branch to the start of the loop for every pass through the loop except the last one. Hence, it is advantageous to assume that this branch will be taken and to have the instruction fetch unit start to fetch instructions at the branch target address. On the other hand, for a branch instruction at the beginning of a program loop, it is advantageous to assume that the branch will not be taken.

A decision on which way to predict the result of the branch may be made in hardware by observing whether the target address of the branch is lower than or higher than the address of the branch instruction. A more flexible approach is to have the compiler decide whether a given branch instruction should be predicted taken or not taken. The branch instructions of some processors, such as SPARC, include a branch prediction bit, which is set to 0 or 1 by the compiler to indicate the desired behavior. The instruction fetch unit checks this bit to predict whether the branch will be taken or not taken.

With either of these schemes, the branch prediction decision is always the same every time a given instruction is executed. Any approach that has this characteristic is called *static branch prediction*. Another approach in which the prediction decision may change depending on execution history is called *dynamic branch prediction*.

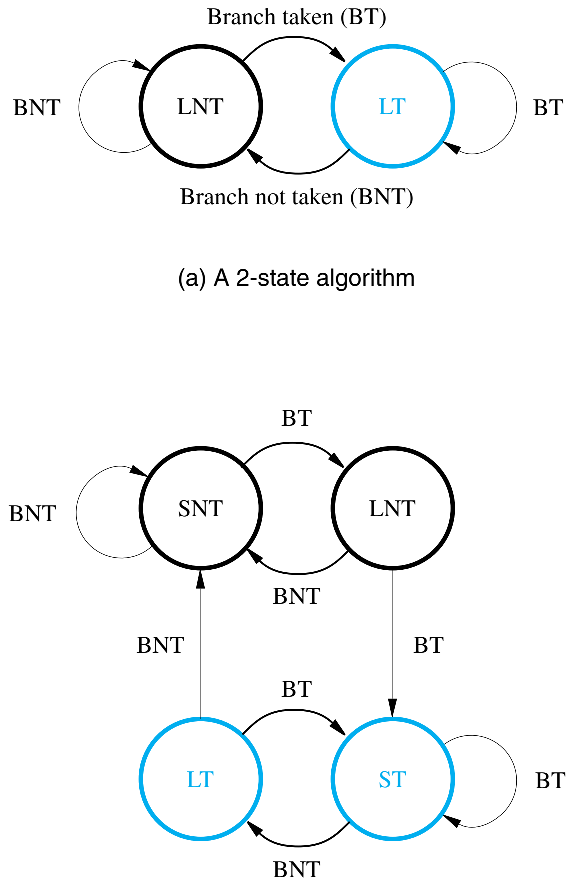
Dynamic Branch Prediction

The objective of branch prediction algorithms is to reduce the probability of making a wrong decision, to avoid fetching instructions that eventually have to be discarded. In dynamic branch prediction schemes, the processor hardware assesses the likelihood of a given branch being taken by keeping track of branch decisions every time that instruction is executed.

In its simplest form, the execution history used in predicting the outcome of a given branch instruction is the result of the most recent execution of that instruction. The processor assumes that the next time the instruction is executed, the result is likely to be the same. Hence, the algorithm may be described by the two-state machine in Figure 8.15*a*. The two states are:

LT:	Branch is likely to be taken
LNT:	Branch is likely not to be taken

Suppose that the algorithm is started in state LNT. When the branch instruction is



(a) A 2-state algorithm

(b) A 4-state algorithm

Figure 8.15 State-machine representation of branch-prediction algorithms.

executed and if the branch is taken, the machine moves to state LT. Otherwise, it remains in state LNT. The next time the same instruction is encountered, the branch is predicted as taken if the corresponding state machine is in state LT. Otherwise it is predicted as not taken.

This simple scheme, which requires one bit of history information for each branch instruction, works well inside program loops. Once a loop is entered, the branch instruction that controls looping will always yield the same result until the last pass through the loop is reached. In the last pass, the branch prediction will turn out to be incorrect, and the branch history state machine will be changed to the opposite state. Unfortunately, this means that the next time this same loop is entered, and assuming that there will be more than one pass through the loop, the machine will lead to the wrong prediction.

Better performance can be achieved by keeping more information about execution history. An algorithm that uses 4 states, thus requiring two bits of history information for each branch instruction, is shown in Figure 8.15*b*. The four states are:

ST:	Strongly likely to be taken
LT:	Likely to be taken
LNT:	Likely not to be taken
SNT:	Strongly likely not to be taken

Again assume that the state of the algorithm is initially set to LNT. After the branch instruction has been executed, and if the branch is actually taken, the state is changed to ST; otherwise, it is changed to SNT. As program execution progresses and the same instruction is encountered again, the state of the branch prediction algorithm continues to change as shown. When a branch instruction is encountered, the instruction fetch unit predicts that the branch will be taken if the state is either LT or ST, and it begins to fetch instructions at the branch target address. Otherwise, it continues to fetch instructions in sequential address order.

It is instructive to examine the behavior of the branch prediction algorithm in some detail. When in state SNT, the instruction fetch unit predicts that the branch will not be taken. If the branch is actually taken, that is if the prediction is incorrect, the state changes to LNT. This means that the next time the same branch instruction is encountered, the instruction fetch unit will still predict that the branch will not be taken. Only if the prediction is incorrect twice in a row will the state change to ST. After that, the branch will be predicted as taken.

Let us reconsider what happens when executing a program loop. Assume that the branch instruction is at the end of the loop and that the processor sets the initial state of the algorithm to LNT. During the first pass, the prediction will be wrong (not taken), and hence the state will be changed to ST. In all subsequent passes the prediction will be correct, except for the last pass. At that time, the state will change to LT. When the loop is entered a second time, the prediction will be correct (branch taken).

We now add one final modification to correct the mispredicted branch at the time the loop is first entered. The cause of the misprediction in this case is the initial state of the branch prediction algorithm. In the absence of additional information about the nature of the branch instruction, we assumed that the processor sets the initial state to LNT. The information needed to set the initial state correctly can be provided by any of the static prediction schemes discussed earlier. Either by comparing addresses or by checking a prediction bit in the instruction, the processor sets the initial state of the algorithm to LNT or LT. In the case of a branch at the end of a loop, the compiler would indicate that the branch should be predicted as taken, causing the initial state to be set to LT. With this modification, branch prediction will be correct all the time, except for the final pass through the loop. Misprediction in this latter case is unavoidable.

The state information used in dynamic branch prediction algorithms may be kept by the processor in a variety of ways. It may be recorded in a look-up table, which is accessed using the low-order part of the branch instruction address. In this case, it is possible for two branch instructions to share the same table entry. This may lead to a branch being mispredicted, but it does not cause an error in execution. Misprediction only introduces a small delay in execution time. An alternative approach is to store the history bits as a tag associated with branch instructions in the instruction cache. We will see in Section 8.7 how this information is handled in the SPARC processor.

8.4 INFLUENCE ON INSTRUCTION SETS

We have seen that some instructions are much better suited to pipelined execution than others. For example, instruction side effects can lead to undesirable data dependencies. In this section, we examine the relationship between pipelined execution and machine instruction features. We discuss two key aspects of machine instructions — addressing modes and condition code flags.

8.4.1 ADDRESSING MODES

Addressing modes should provide the means for accessing a variety of data structures simply and efficiently. Useful addressing modes include index, indirect, autoincrement, and autodecrement. Many processors provide various combinations of these modes to increase the flexibility of their instruction sets. Complex addressing modes, such as those involving double indexing, are often encountered.

In choosing the addressing modes to be implemented in a pipelined processor, we must consider the effect of each addressing mode on instruction flow in the pipeline. Two important considerations in this regard are the side effects of modes such as autoincrement and autodecrement and the extent to which complex addressing modes cause the pipeline to stall. Another important factor is whether a given mode is likely to be used by compilers.

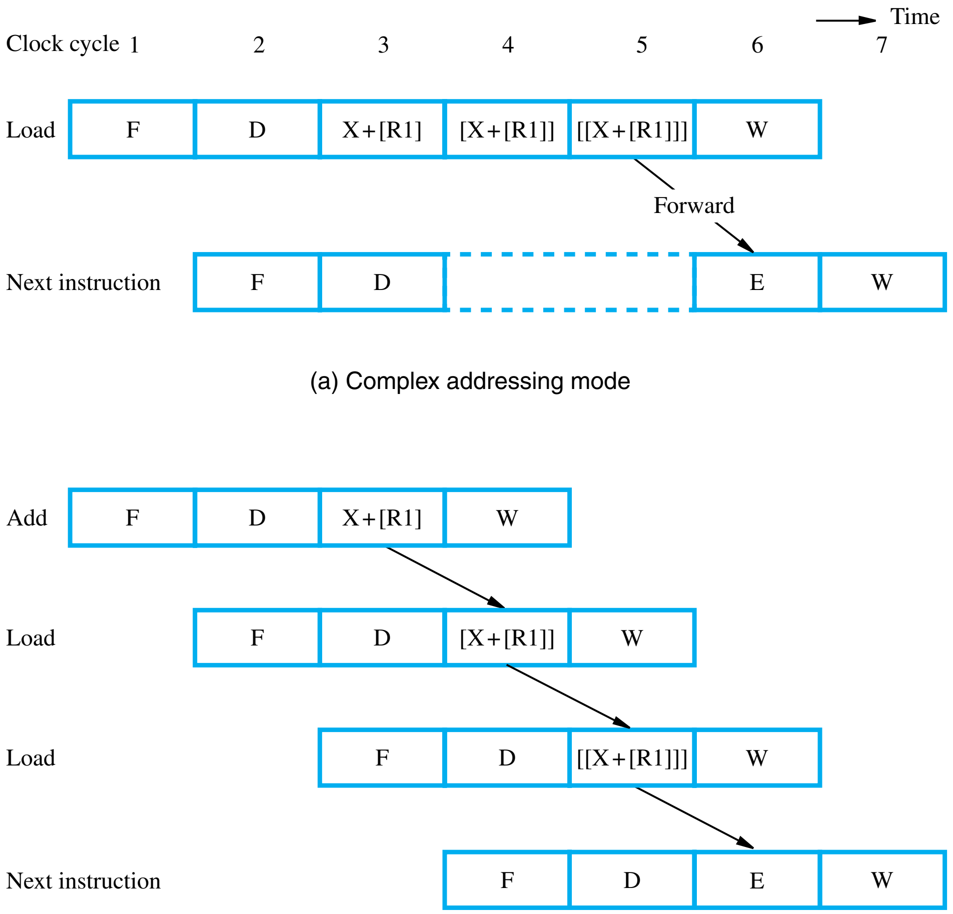
To compare various approaches, we assume a simple model for accessing operands in the memory. The load instruction Load X(R1),R2 takes five cycles to complete execution, as indicated in Figure 8.5. However, the instruction

```
Load    (R1),R2
```

can be organized to fit a four-stage pipeline because no address computation is required. Access to memory can take place in stage E. A more complex addressing mode may require several accesses to the memory to reach the named operand. For example, the instruction

```
Load    (X(R1)),R2
```

may be executed as shown in Figure 8.16*a*, assuming that the index offset, X, is given in the instruction word. After computing the address in cycle 3, the processor needs to access memory twice — first to read location X+[R1] in clock cycle 4 and then to read location [X+[R1]] in cycle 5. If R2 is a source operand in the next instruction, that instruction would be stalled for three cycles, which can be reduced to two cycles with operand forwarding, as shown.



(a) Complex addressing mode

(b) Simple addressing mode

Figure 8.16 Equivalent operations using complex and simple addressing modes.

To implement the same Load operation using only simple addressing modes requires several instructions. For example, on a computer that allows three operand addresses, we can use

```
Add    #X,R1,R2
Load   (R2),R2
Load   (R2),R2
```

The Add instruction performs the operation R2 ← X+ [R1]. The two Load instructions fetch the address and then the operand from the memory. This sequence of instructions takes exactly the same number of clock cycles as the original, single Load instruction, as shown in Figure 8.16*b*.

This example indicates that, in a pipelined processor, complex addressing modes that involve several accesses to the memory do not necessarily lead to faster execution. The main advantage of such modes is that they reduce the number of instructions needed to perform a given task and thereby reduce the program space needed in the main memory. Their main disadvantage is that their long execution times cause the pipeline to stall, thus reducing its effectiveness. They require more complex hardware to decode and execute them. Also, they are not convenient for compilers to work with.

The instruction sets of modern processors are designed to take maximum advantage of pipelined hardware. Because complex addressing modes are not suitable for pipelined execution, they should be avoided. The addressing modes used in modern processors often have the following features:

- Access to an operand does not require more than one access to the memory.
- Only load and store instructions access memory operands.
- The addressing modes used do not have side effects.

Three basic addressing modes that have these features are register, register indirect, and index. The first two require no address computation. In the index mode, the address can be computed in one cycle, whether the index value is given in the instruction or in a register. Memory is accessed in the following cycle. None of these modes has any side effects, with one possible exception. Some architectures, such as ARM, allow the address computed in the index mode to be written back into the index register. This is a side effect that would not be allowed under the guidelines above. Note also that relative addressing can be used; this is a special case of indexed addressing in which the program counter is used as the index register.

The three features just listed were first emphasized as part of the concept of RISC processors. The SPARC processor architecture, which adheres to these guidelines, is presented in Section 8.7.

8.4.2 CONDITION CODES

In many processors, such as those described in Chapter 3, the condition code flags are stored in the processor status register. They are either set or cleared by many instructions, so that they can be tested by subsequent conditional branch instructions to change the flow of program execution. An optimizing compiler for a pipelined processor attempts to reorder instructions to avoid stalling the pipeline when branches or data dependencies between successive instructions occur. In doing so, the compiler must ensure that reordering does not cause a change in the outcome of a computation. The dependency introduced by the condition-code flags reduces the flexibility available for the compiler to reorder instructions.

Consider the sequence of instructions in Figure 8.17*a*, and assume that the execution of the Compare and Branch=0 instructions proceeds as in Figure 8.14. The branch decision takes place in step E_2 rather than D_2 because it must await the result of the Compare instruction. The execution time of the Branch instruction can be reduced

```
Add             R1,R2
Compare         R3,R4
Branch=0        . . .
```

(a) A program fragment

```
Compare         R3,R4
Add             R1,R2
Branch=0        . . .
```

(b) Instructions reordered

Figure 8.17 Instruction reordering.

by interchanging the Add and Compare instructions, as shown in Figure 8.17*b*. This will delay the branch instruction by one cycle relative to the Compare instruction. As a result, at the time the Branch instruction is being decoded the result of the Compare instruction will be available and a correct branch decision will be made. There would be no need for branch prediction. However, interchanging the Add and Compare instructions can be done only if the Add instruction does not affect the condition codes.

These observations lead to two important conclusions about the way condition codes should be handled. First, to provide flexibility in reordering instructions, the condition-code flags should be affected by as few instructions as possible. Second, the compiler should be able to specify in which instructions of a program the condition codes are affected and in which they are not. An instruction set designed with pipelining in mind usually provides the desired flexibility. Figure 8.17*b* shows the instructions reordered assuming that the condition code flags are affected only when this is explicitly stated as part of the instruction OP code. The SPARC and ARM architectures provide this flexibility.

8.5 DATAPATH AND CONTROL CONSIDERATIONS

Organization of the internal datapath of a processor was introduced in Chapter 7. Consider the three-bus structure presented in Figure 7.8. To make it suitable for pipelined execution, it can be modified as shown in Figure 8.18 to support a 4-stage pipeline. The resources involved in stages F and E are shown in blue and those used in stages D and W in black. Operations in the data cache may happen during stage E or at a later stage, depending on the addressing mode and the implementation details. This section

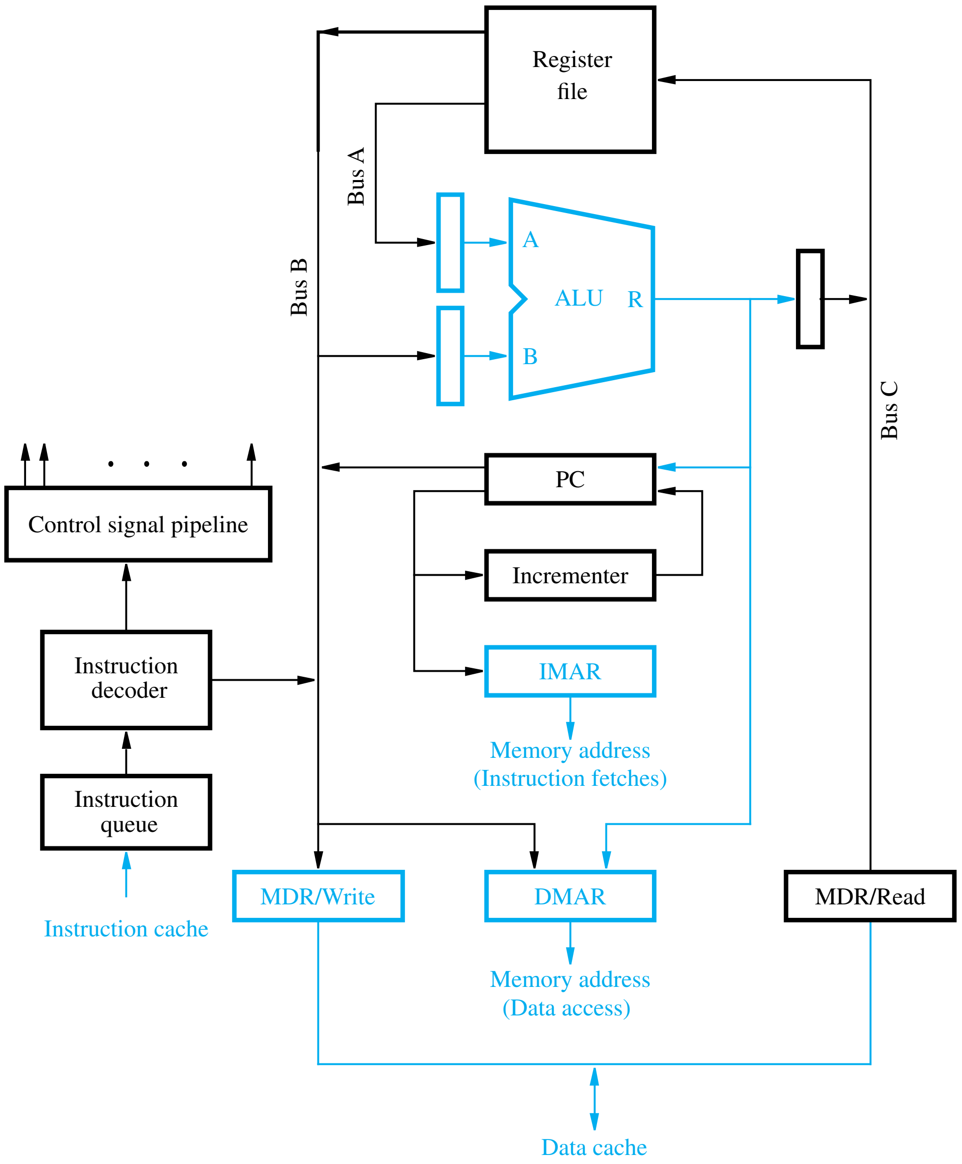


Figure 8.18 Datapath modified for pipelined execution with interstage buffers at the input and output of the ALU.

is shown in blue. Several important changes to Figure 7.8 should be noted:

1. There are separate instruction and data caches that use separate address and data connections to the processor. This requires two versions of the MAR register, IMAR for accessing the instruction cache and DMAR for accessing the data cache.

2. The PC is connected directly to the IMAR, so that the contents of the PC can be transferred to IMAR at the same time that an independent ALU operation is taking place.

3. The data address in DMAR can be obtained directly from the register file or from the ALU to support the register indirect and indexed addressing modes.

4. Separate MDR registers are provided for read and write operations. Data can be transferred directly between these registers and the register file during load and store operations without the need to pass through the ALU.

5. Buffer registers have been introduced at the inputs and output of the ALU. These are registers SRC1, SRC2, and RSLT in Figure 8.7. Forwarding connections are not included in Figure 8.18. They may be added if desired.

6. The instruction register has been replaced with an instruction queue, which is loaded from the instruction cache.

7. The output of the instruction decoder is connected to the control signal pipeline. The need for buffering control signals and passing them from one stage to the next along with the instruction is discussed in Section 8.1. This pipeline holds the control signals in buffers B2 and B3 in Figure 8.2*a*.

The following operations can be performed independently in the processor of Figure 8.18:

- Reading an instruction from the instruction cache
- Incrementing the PC
- Decoding an instruction
- Reading from or writing into the data cache
- Reading the contents of up to two registers from the register file
- Writing into one register in the register file
- Performing an ALU operation

Because these operations do not use any shared resources, they can be performed simultaneously in any combination. The structure provides the flexibility required to implement the four-stage pipeline in Figure 8.2. For example, let I_1, I_2, I_3, and I_4 be a sequence of four instructions. As shown in Figure 8.2*a*, the following actions all happen during clock cycle 4:

- Write the result of instruction I_1 into the register file
- Read the operands of instruction I_2 from the register file
- Decode instruction I_3
- Fetch instruction I_4 and increment the PC.

8.6 SUPERSCALAR OPERATION

Pipelining makes it possible to execute instructions concurrently. Several instructions are present in the pipeline at the same time, but they are in different stages of their execution. While one instruction is performing an ALU operation, another instruction is being decoded and yet another is being fetched from the memory. Instructions enter the pipeline in strict program order. In the absence of hazards, one instruction enters the pipeline and one instruction completes execution in each clock cycle. This means that the maximum throughput of a pipelined processor is one instruction per clock cycle.

A more aggressive approach is to equip the processor with multiple processing units to handle several instructions in parallel in each processing stage. With this arrangement, several instructions start execution in the same clock cycle, and the processor is said to use *multiple-issue*. Such processors are capable of achieving an instruction execution throughput of more than one instruction per cycle. They are known as *superscalar* processors. Many modern high-performance processors use this approach.

We introduced the idea of an instruction queue in Section 8.3. We pointed out that to keep the instruction queue filled, a processor should be able to fetch more than one instruction at a time from the cache. For superscalar operation, this arrangement is essential. Multiple-issue operation requires a wider path to the cache and multiple execution units. Separate execution units are provided for integer and floating-point instructions.

Figure 8.19 shows an example of a processor with two execution units, one for integer and one for floating-point operations. The Instruction fetch unit is capable of reading two instructions at a time and storing them in the instruction queue. In each clock cycle, the Dispatch unit retrieves and decodes up to two instructions from the front of the queue. If there is one integer, one floating-point instruction, and no hazards, both instructions are dispatched in the same clock cycle.

In a superscalar processor, the detrimental effect on performance of various hazards becomes even more pronounced. The compiler can avoid many hazards through judicious selection and ordering of instructions. For example, for the processor in Figure 8.19, the compiler should strive to interleave floating-point and integer instructions. This would enable the dispatch unit to keep both the integer and floating-point

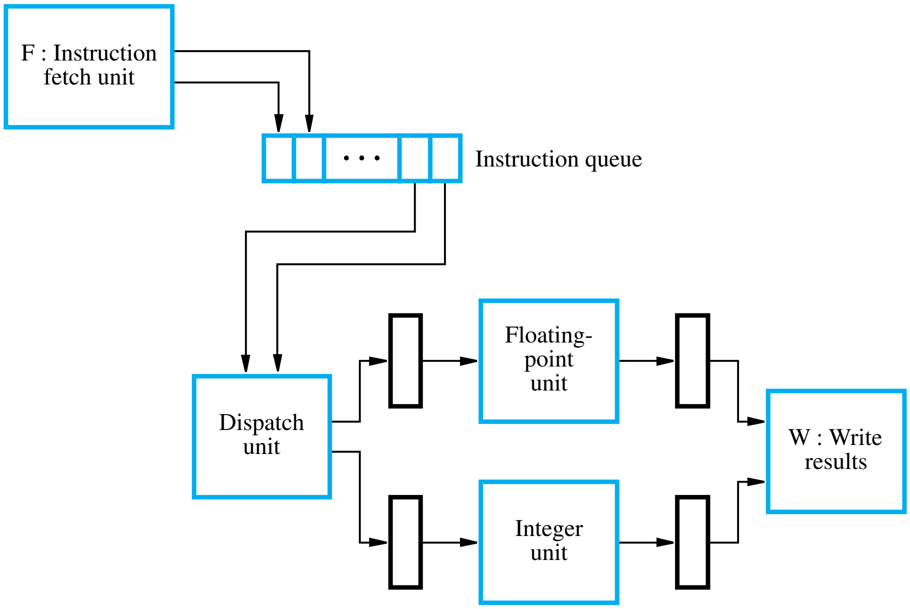


Figure 8.19 A processor with two execution units.

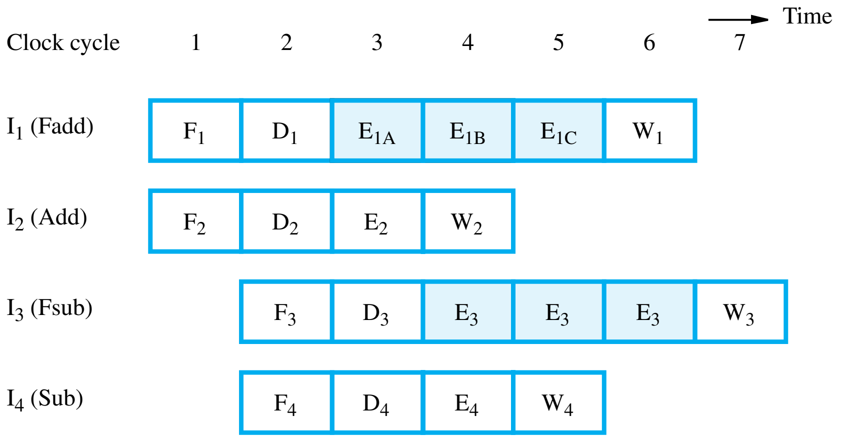


Figure 8.20 An example of instruction execution flow in the processor of Figure 8.19, assuming no hazards are encountered.

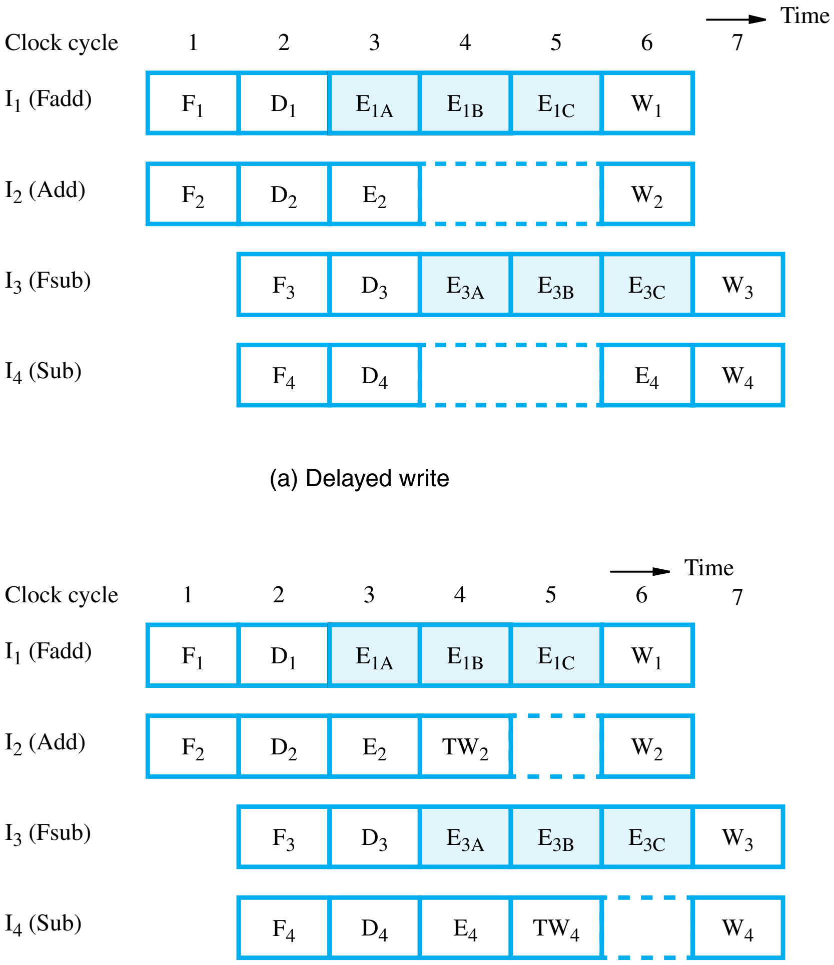
units busy most of the time. In general, high performance is achieved if the compiler is able to arrange program instructions to take maximum advantage of the available hardware units.

Pipeline timing is shown in Figure 8.20. The blue shading indicates operations in the floating-point unit. The floating-point unit takes three clock cycles to complete the floating-point operation specified in I_1. The integer unit completes execution of I_2 in one clock cycle. We have also assumed that the floating-point unit is organized internally as a three-stage pipeline. Thus, it can still accept a new instruction in each clock cycle. Hence, instructions I_3 and I_4 enter the dispatch unit in cycle 3, and both are dispatched in cycle 4. The integer unit can receive a new instruction because instruction I_2 has proceeded to the Write stage. Instruction I_1 is still in the execution phase, but it has moved to the second stage of the internal pipeline in the floating-point unit. Therefore, instruction I_3 can enter the first stage. Assuming that no hazards are encountered, the instructions complete execution as shown.

8.6.1 OUT-OF-ORDER EXECUTION

In Figure 8.20, instructions are dispatched in the same order as they appear in the program. However, their execution is completed out of order. Does this lead to any problems? We have already discussed the issues arising from dependencies among instructions. For example, if instruction I_2 depends on the result of I_1, the execution of I_2 will be delayed. As long as such dependencies are handled correctly, there is no reason to delay the execution of an instruction. However, a new complication arises when we consider the possibility of an instruction causing an exception. Exceptions may be caused by a bus error during an operand fetch or by an illegal operation, such as an attempt to divide by zero. The results of I_2 are written back into the register file in cycle 4. If instruction I_1 causes an exception, program execution is in an inconsistent state. The program counter points to the instruction in which the exception occurred. However, one or more of the succeeding instructions have been executed to completion. If such a situation is permitted, the processor is said to have *imprecise exceptions*.

To guarantee a consistent state when exceptions occur, the results of the execution of instructions must be written into the destination locations strictly in program order. This means we must delay step W_2 in Figure 8.20 until cycle 6. In turn, the integer execution unit must retain the result of instruction I_2, and hence it cannot accept instruction I_4 until cycle 6, as shown in Figure 8.21*a*. If an exception occurs during an instruction,



(a) Delayed write

(b) Using temporary registers

Figure 8.21 Instruction completion in program order.

all subsequent instructions that may have been partially executed are discarded. This is called a *precise exception.*

It is easier to provide precise exceptions in the case of external interrupts. When an external interrupt is received, the Dispatch unit stops reading new instructions from the instruction queue, and the instructions remaining in the queue are discarded. All instructions whose execution is pending continue to completion. At this point, the processor and all its registers are in a consistent state, and interrupt processing can begin.

8.6.2 EXECUTION COMPLETION

It is desirable to use out-of-order execution, so that an execution unit is freed to execute other instructions as soon as possible. At the same time, instructions must be completed in program order to allow precise exceptions. These seemingly conflicting requirements are readily resolved if execution is allowed to proceed as shown in Figure 8.20, but the results are written into temporary registers. The contents of these registers are later transferred to the permanent registers in correct program order. This approach is illustrated in Figure 8.21*b*. Step TW is a write into a temporary register. Step W is the final step in which the contents of the temporary register are transferred into the appropriate permanent register. This step is often called the *commitment* step because the effect of the instruction cannot be reversed after that point. If an instruction causes an exception, the results of any subsequent instruction that has been executed would still be in temporary registers and can be safely discarded.

A temporary register assumes the role of the permanent register whose data it is holding and is given the same name. For example, if the destination register of I_2 is R5, the temporary register used in step TW_2 is treated as R5 during clock cycles 6 and 7. Its contents would be forwarded to any subsequent instruction that refers to R5 during that period. Because of this feature, this technique is called *register renaming*. Note that the temporary register is used only for instructions that *follow* I_2 in program order. If an instruction that precedes I_2 needs to read R5 in cycle 6 or 7, it would access the actual register R5, which still contains data that have not been modified by instruction I_2.

When out-of-order execution is allowed, a special control unit is needed to guarantee in-order commitment. This is called the *commitment unit*. It uses a queue called the *reorder buffer* to determine which instruction(s) should be committed next. Instructions are entered in the queue strictly in program order as they are dispatched for execution. When an instruction reaches the head of that queue and the execution of that instruction has been completed, the corresponding results are transferred from the temporary registers to the permanent registers and the instruction is removed from the queue. All resources that were assigned to the instruction, including the temporary registers, are released. The instruction is said to have been *retired* at this point. Because an instruction is retired only when it is at the head of the queue, all instructions that were dispatched before it must also have been retired. Hence, instructions may complete execution out of order, but they are retired in program order.

8.6.3 DISPATCH OPERATION

We now return to the dispatch operation. When dispatching decisions are made, the dispatch unit must ensure that all the resources needed for the execution of an instruction are available. For example, since the results of an instruction may have to be written in a temporary register, the required register must be free, and it is reserved for use by that instruction as a part of the dispatch operation. A location in the reorder buffer must also be available for the instruction. When all the resources needed are assigned, including an appropriate execution unit, the instruction is dispatched.

Should instructions be dispatched out of order? For example, if instruction I_2 in Figure 8.20*b* is delayed because of a cache miss for a source operand, the integer unit will be busy in cycle 4, and I_4 cannot be dispatched. Should I_5 be dispatched instead? In principle this is possible, provided that a place is reserved in the reorder buffer for instruction I_4 to ensure that all instructions are retired in the correct order. Dispatching instructions out of order requires considerable care. If I_5 is dispatched while I_4 is still waiting for some resource, we must ensure that there is no possibility of a deadlock occurring.

A *deadlock* is a situation that can arise when two units, A and B, use a shared resource. Suppose that unit B cannot complete its task until unit A completes its task. At the same time, unit B has been assigned a resource that unit A needs. If this happens, neither unit can complete its task. Unit A is waiting for the resource it needs, which is being held by unit B. At the same time, unit B is waiting for unit A to finish before it can release that resource.

If instructions are dispatched out of order, a deadlock can arise as follows. Suppose that the processor has only one temporary register, and that when I_5 is dispatched, that register is reserved for it. Instruction I_4 cannot be dispatched because it is waiting for the temporary register, which, in turn, will not become free until instruction I_5 is retired. Since instruction I_5 cannot be retired before I_4, we have a deadlock.

To prevent deadlocks, the dispatcher must take many factors into account. Hence, issuing instructions out of order is likely to increase the complexity of the Dispatch unit significantly. It may also mean that more time is required to make dispatching decisions. For these reasons, most processors use only in-order dispatching. Thus, the program order of instructions is enforced at the time instructions are dispatched and again at the time instructions are retired. Between these two events, the execution of several instructions can proceed at their own speed, subject only to any interdependencies that may exist among instructions.

In the next section, we present the UltraSPARC II as a case study of a commercially successful, superscalar, highly pipelined processor. The way in which the various issues raised in this chapter have been handled in this processor and the choices made are highly instructive.

8.7 UltraSPARC II EXAMPLE

Processor design has advanced greatly in recent years. The classification of processors as either purely RISC or CISC is no longer appropriate because modern high-performance processors contain elements of both design styles.

The early RISC processors showed how certain features can contribute to high performance. The following two observations proved to be particularly important:

- Pipelining, which enables a processor to execute several instructions at the same time, can lead to significant performance enhancements provided that the pipeline is not stalled frequently.
- A close synergy between the hardware and compiler design enables the compiler to take maximum advantage of the pipelined structure by reducing the events that lead to pipeline stalls.

It is these factors, rather than simply a reduced instruction set, that have contributed to the success of RISC processors. Of particular importance in this regard is the close coordination between the design of the hardware, particularly the structure of the pipeline, and the compiler. Much of the credit for today's high levels of performance goes to developments in compiler technology, which in turn have led to new hardware features that would have been of little use a few years ago.

The SPARC architecture, which is the basis for the processors used in Sun workstations, is an excellent case in point. One of Sun's implementations of the SPARC architecture is called UltraSPARC II. This is the processor we will discuss. We have chosen it instead of one of the processors presented in Chapter 3 because it illustrates very well superscalar operation as well as most of the pipeline design options and trade-offs discussed in this chapter. We will start with a brief introduction to the SPARC architecture. For a complete description, the reader should consult the SPARC Architecture Manual [1].

8.7.1 SPARC Architecture

SPARC stands for Scalable Processor ARChitecture. It is a specification of the instruction set architecture of a processor, that is, it is a specification of the processor's instruction set and register organization, regardless of how these may be implemented in hardware. Furthermore, SPARC is an "open architecture," which means that computer companies other than Sun Microsystems can develop their own hardware to implement the same instruction set.

The SPARC architecture was first announced in 1987, based on ideas developed at the University of California at Berkeley in the early eighties, in a project that coined the name reduced instruction set computer and its corresponding acronym RISC. The Sun Corporation and several other processor chip manufacturers have designed and built many processors based on this architecture, covering a wide range of performance. The SPARC architecture specifications are controlled by an international consortium, which introduces new enhanced versions every few years. The most recent version is SPARC-V9.

The instruction set of the SPARC architecture has a distinct RISC style. The architecture specifications describe a processor in which data and memory addresses are 64 bits long. Instructions are of equal length, and they are all 32 bits long. Both integer and floating-point instructions are provided.

There are two register files, one for integer data and one for floating-point data. Integer registers are 64 bits long. Their number is implementation dependent and can vary from 64 to 528. SPARC uses a scheme known as *register windows*. At any given time, an application program sees only 32 registers, called R0 to R31. Of these, the first eight are global registers that are always accessible. The remaining 24 registers are local to the current context.

Floating-point registers are only 32 bits long because this is the length of single-precision floating-point numbers according to the IEEE Standard described in Chapter 6. The instruction set includes floating-point instructions for double- and quad-precision operations. Two sequentially numbered floating-point registers are used to hold a double-precision operand and four are used for quad precision. There is a total of 64 registers, F0 to F63. Single precision operands can be stored in F0 to F31, double precision operands in F0, F2, F4, . . . , F62, and quad-precision in F0, F4, F8, . . . , F60.

Load and Store Instructions

Only load and store instructions access the memory, where an operand may be an 8-bit byte, a 16-bit half word, or a 32-bit word. Load and store instructions also handle 64-bit quantities, which come in two varieties: extended word or doubleword. An LDX (Load extended) instruction loads a 64-bit quantity, called an *extended* word, into one of the processor's integer registers. A *doubleword* consists of two 32-bit words. The two words are loaded into two sequentially numbered processor registers using a single LDD (Load double) instruction. They are loaded into the low-order 32 bits of each register, and the high order bits are filled with 0s. The first of the two registers, which is the register named in the instruction, must be even numbered. Load and store instructions that handle doublewords are useful for moving multiple-precision floating-point operands between the memory and floating-point registers.

Load and store instructions use one of two indexed addressing modes, as follows:

1. The effective address is the sum of the contents of two registers:

$$\text{EA} = [\text{Radr1}] + [\text{Radr2}]$$

2. The effective address is the sum of the contents of one register plus an immediate operand that is included in the instruction

$$\text{EA} = [\text{Radr1}] + \text{Immediate}$$

For most instructions, the immediate operand is a signed 13-bit value. It is sign-extended to 64 bits and then added to the contents of Radr1.

A load instruction that uses the first addressing mode is written as

```
Load    [Radr1+Radr2], Rdst
```

It generates the effective address [Radr1] + [Radr2] and loads the contents of that location into register Rdst. For an immediate displacement, Radr2 is replaced with the immediate operand value, which yields

Load [Radr1+Imm], Rdst

Store instructions use a similar syntax, with the first operand specifying the source register from which data will be stored in the memory, as follows:

Store Rsrc, [Radr1+Radr2]

Store Rsrc, [Radr1+Imm]

In the recommended syntax for SPARC instructions, a register is specified by a % sign followed by the register number. Either %r2 or %2 refers to register number 2. However, for better readability and consistency with earlier chapters, we will use R0, R1, and so on, to refer to integer registers and F0, F1, . . . for floating-point registers.

As an example, consider the Load unsigned byte instruction

LDUB [R2+R3], R4

This instruction loads one byte from memory location [R2] + [R3] into the low-order 8 bits of register R4, and fills the high-order 56 bits with 0s. The Load signed word instruction:

LDSW [R2+2500], R4

reads a 32-bit word from location [R2] + 2500, sign extends it to 64 bits, and then stores it in register R4.

Arithmetic and Logic Instructions

The usual set of arithmetic and logic instructions is provided. A few examples are shown in Table 8.1. We pointed out in Section 8.4.2 that an instruction should set the condition code flags only when these flags are going to be tested by a subsequent conditional branch instruction. This maximizes the flexibility the compiler has in reordering instructions to avoid stalling the pipeline. The SPARC instruction set has been designed with this feature in mind. Arithmetic and logic instructions are available in two versions, one sets the condition code flags and the other does not. The suffix cc in an OP code is used to indicate that the flags should be set. For example, the instructions ADD, SUB, SMUL (signed multiply), OR, and XOR do not affect the flags, while ADDcc and SUBcc do.

Register R0 always contains the value 0. When it is used as the destination operand, the result of the instruction is discarded. For example, the instruction

SUBcc R2, R3, R0

subtracts the contents of R3 from R2, sets the condition code flags, and discards the result of the subtraction operation. In effect, this is a compare instruction, and it has the alternative syntax

CMP R2, R3

In the SPARC nomenclature, CMP is called a synthetic instruction. It is not a real

Table 8.1 Examples of SPARC instructions

Instruction		Description
ADD	R5, R6, R7	Integer add: R7 ← [R5] + [R6]
ADDcc	R2, R3, R5	R5 ← [R2] + [R3], set condition code flags
SUB	R5, Imm, R7	Integer subtract: R7 ← [R5] − Imm(sign-extended)
AND	R3, Imm, R5	Bitwise AND: R5 ← [R3] AND Imm(sign-extended)
XOR	R3, R4, R5	Bitwise Exclusive OR: R5 ← [R3] XOR [R4]
FADDq	F4, F12, F16	Floating-point add, quad precision: F12 ← [F4] + [F12]
FSUBs	F2, F5, F7	Floating-point subtract, single precision: F7 ← [F2] − [F5]
FDIVs	F5, F10, F18	Floating-point divide, single precision, F18 ← [F5]/[F10]
LDSW	R3, R5, R7	R7 ← 32-bit word at [R3] + [R5] sign extended to a 64-bit value
LDX	R3, R5, R7	R7 ← 64-bit extended word at [R3] + [R5]
LDUB	R4, Imm, R5	Load unsigned byte from memory location [R4] + Imm, the byte is loaded into the least significant 8 bits of register R5, and all higher-order bits are filled with 0s
STW	R3, R6, R12	Store word from register R3 into memory location [R6] + [R12]
LDF	R5, R6, F3	Load a 32-bit word at address [R5] + [R6] into floating-point register F3
LDDF	R5, R6, F8	Load doubleword (two 32-bit words) at address [R5] + [R6] into floating-point registers F8 and F9
STF	F14, R6, Imm	Store word from floating-register F14 into memory location [R6] + Imm
BLE	icc, Label	Test the icc flags and branch to Label if less than or equal to zero
BZ,pn	xcc, Label	Test the xcc flags and branch to Label if equal to zero, branch is predicted not taken
BGT,a,pt	icc, Label	Test the 32-bit integer condition codes and branch to Label if greater than zero, set annul bit, branch is predicted taken
FBNE,pn	Label	Test floating-point status flags and branch if not equal, the annul bit is set to zero, and the branch is predicted not taken

instruction recognized by the hardware. It is provided only for the convenience of the programmer. The assembler replaces it with a SUBcc instruction.

A condition code register, CCR, is provided, which contains two sets of condition code flags, *icc* and *xcc,* for integer and extended condition codes, respectively. Each set consists of four flags N, Z, V, and C. Instructions that set the condition code flags, such as ADDcc, will set both the *icc* and *xcc* bits; the *xcc* flags are set based on the 64-bit result of the instruction, and the *icc* flags are set based on the low-order 32 bits only.

The condition codes for floating-point operations are held in a 64-bit register called the floating-point state register, FSR.

Branch Instructions

The way in which branches are handled is an important factor in determining performance. Branch instructions in the SPARC instruction set contain several features that are intended to enhance performance of a pipelined processor and to help the compiler in optimizing the code it emits.

A SPARC processor uses delayed branching with one delay slot (see Section 8.3.2). Branch instructions include a branch prediction bit, which the compiler can use to give the hardware a hint about the expected behavior of the branch. Branch instructions also contain an Annul bit, which is intended to increase flexibility in handling the instruction in the delay slot. This instruction is always executed, but its results are not committed until after the branch decision is known. If the branch is taken, execution of the instruction in the delay slot is completed and the results are committed. If the branch is not taken, this instruction is annulled if the Annul bit is equal to 1. Otherwise, execution of the instruction is completed.

The compiler may be able to place in the delay slot an instruction that is needed whether or not the branch is taken. This may be an instruction that logically belongs before the branch instruction but can be moved into the delay slot. The Annul bit should be set to 0 in this case. Otherwise, the delay slot should be filled with an instruction that is to be executed only if the branch is taken, in which case the Annul bit should be set to 1.

Conditional branch instructions can test the *icc, xcc,* or FSR flags. For example, the instruction

```
BGT,a,pt    icc, Label
```

will cause a branch to location Label if the previous instruction that set the flags in *icc* produced a greater-than-zero result. The instruction will have both the Annul bit and the branch prediction bit set to 1. The instruction

```
FBGT,a,pt    Label
```

is exactly the same, except that it will test the FSR flags. If neither pt (predicted taken) nor pn (predicted not taken) is specified, the assembler will default to pt.

An example that illustrates the prediction and annul facilities in branch instructions is given in Figure 8.22, which shows a program loop that adds a list of *n* 64-bit integers. We have assumed that the number of items in the list is stored at address LIST as a 64-bit integer, followed by the numbers to be added in successive 64-bit locations. We have also assumed that there is at least one item in the list and that the address LIST has been loaded into register R3 earlier in the program.

Figure 8.22*a* shows the desired loop as it would be written for execution on a nonpipelined processor. For execution on a SPARC processor, we should first reorganize the instructions to make effective use of the branch delay slot. Observe that the ADD instruction following LOOPSTART is executed during every pass through the loop.

```
            LDX     R3, 0, R6          Load number of items in the list.
            OR      R0, R0, R4         R4 to be used as offset in the list
            OR      R0, R0, R7         Clear R7 to be used as accumulator.
LOOPSTART   LDX     R3, R4, R5         Load list item into R5.
            ADD     R5, R7, R7         Add number to accumulator.
            ADD     R4, 8, R4          Point to the next entry.
            SUBcc   R6, 1, R6          Decrement R6 and set condition flags.
            BG      xcc, LOOPSTART     Loop if more items in the list.
NEXT        ...
```

(a) Desired program loop

```
            LDX     R3, 0, R6
            OR      R0, R0, R4
            OR      R0, R0, R7
LOOPSTART   LDX     R3, R4, R5
            ADD     R4, 8, R4
            SUBcc   R6, 1, R6
            BG,pt   xcc, LOOPSTART     Predicted taken, Annul bit = 0
            ADD     R5, R7, R7
NEXT        ...
```

(b) Instructions reorganized to use the delay slot

Figure 8.22 An addition loop showing the use of the branch delay slot and branch prediction.

Also, none of the instructions following it depends on its result. Hence, this instruction may be moved into the delay slot following the branch at the end of the loop, as shown in Figure 8.22*b*. Since it is to be executed regardless of the branch outcome, the Annul bit in the branch instruction is set to 0 (this is the default condition).

As for branch prediction, observe that the number of times the loop will be executed is equal to the number of items in the list. This means that, except for the trivial case of $n = 1$, the branch will be taken a number of times before exiting the loop. Hence, we have set the branch prediction bit in the BG instruction to indicate that the branch is expected to be taken.

Conditional branch instructions are not the only instructions that check the condition code flags. For example, there is a conditional move instruction, MOVcc, which copies data from one register into another only if the condition codes satisfy the condition specified in the instruction suffix, cc. Consider the two instructions

```
CMP     R5, R6
MOVle   icc, R5, R6
```

The MOVle instruction copies the contents of R5 into R6 if the condition code flags in *icc* indicate a less-than-or-equal-to condition $(Z + (N \oplus V) = 1)$. The net result is to place the smaller of the two values in register R6. In the absence of a conditional move instruction, the same task would require a branch instruction, as in the following sequence

```
          CMP    R5, R6
          BG     icc, GREATER
          MOVA   icc, R5, R6
GREATER   . . .
```

where MOVA is the move-always instruction. The MOVle instruction not only reduces the number of instructions needed, but more importantly, it avoids the performance degradation caused by branch instructions in pipelined execution.

The instruction set has many other features that are intended to maximize performance in a highly pipelined superscalar processor. We will discuss some of these features in the context of the UltraSPARC II processor. The ideas behind these features have already been introduced earlier in the chapter.

8.7.2 UltraSPARC II

The main building blocks of the UltraSPARC II processor are shown in Figure 8.23. The processor uses two levels of cache: an external cache (E-cache) and two internal caches, one for instructions (I-cache) and one for data (D-cache). The external cache controller is on the processor chip, as is the control hardware for memory management. The memory management unit uses two translation lookaside buffers, one for instructions, iTLB, and one for data, dTLB. The processor communicates with the memory and the I/O subsystem over the system interconnection bus.

There are two execution units, one for integer and one for floating-point operations. Each of these units contains a register set and two independent pipelines for instruction execution. Thus, the processor can simultaneously start the execution of up to four instructions, two integer and two floating-point. These four instructions proceed in parallel, each through its own pipeline. If instructions are available and none of the four pipelines is stalled, four new instructions can enter the execution phase every clock cycle.

The Prefetch and Dispatch Unit (PDU) of the processor is responsible for maintaining a continuous supply of instructions for the execution units. It does so by prefetching instructions before they are needed and placing them in a temporary storage buffer called the instruction buffer, which performs the role of the instruction queue in Figure 8.19.

8.7.3 Pipeline Structure

The UltraSPARC II has a nine-stage instruction execution pipeline, shown in Figure 8.24. The function of each stage is completed in one processor clock cycle. We will give an overview of the operation of the pipeline, then discuss each stage in detail.

The first three stages of the pipeline are common to all instructions. Instructions

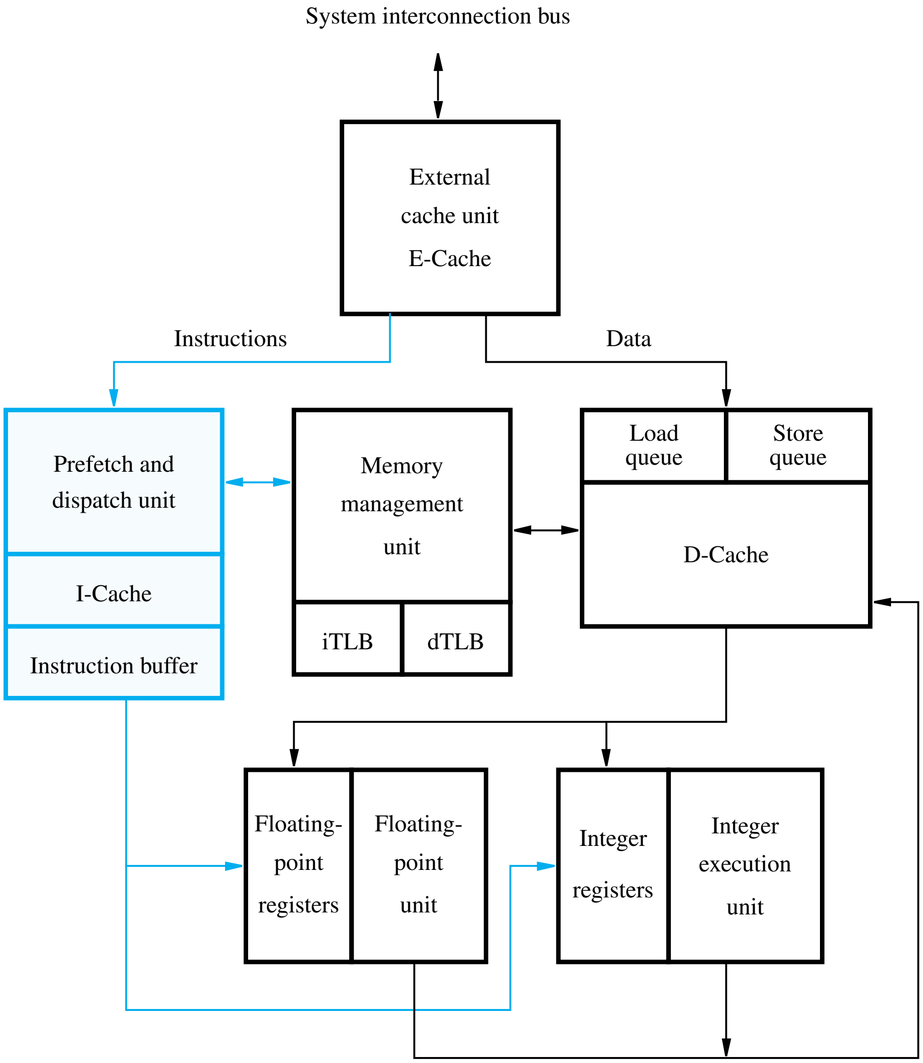


Figure 8.23 Main building blocks of the UltraSPARC II processor.

are fetched from the instruction cache in the first stage (F) and partially decoded in the second stage (D). Then, in the third stage (G), a group of up to four instructions is selected for execution in parallel. The instructions are then dispatched to the integer and floating-point execution units.

Each of the two execution units consists of two parallel pipelines with six stages each. The first four stages are available to perform the operation specified by the instruction, and the last two are used to check for exceptions and store the result of the instruction.

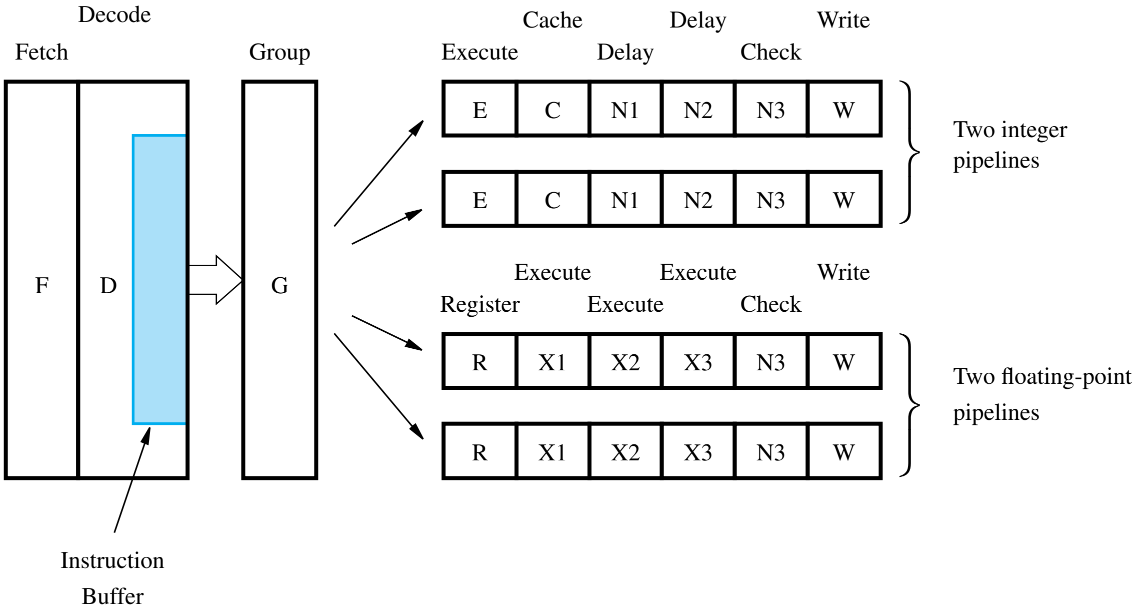


Figure 8.24 Pipeline organization of the UltraSPARC II processor.

Instruction Fetch and Decode

The PDU fetches up to four instructions from the instruction cache, partially decodes them, and stores the results in the instruction buffer, which can hold up to 12 instructions. The decoding that takes place in this stage enables the PDU to determine whether the instruction is a branch instruction. It also detects salient features that can be used to speed up the decisions to be made later in the pipeline.

A cache block in the instruction cache consists of 32 bytes. It contains eight instructions. As instructions are loaded into the cache they are stored based on their virtual addresses, so that they can be fetched quickly by the PDU without requiring address translation. The PDU can maintain the rate of four instructions per cycle as long as each group does not cross cache block boundaries. If there are fewer than four instructions left in a cache block, the unit will read only the remaining instructions in the current block.

The PDU uses a four-state branch prediction algorithm similar to that described in Figure 8.15. It uses the branch prediction bit in the branch instruction to set the initial state to either LT or LNT. For every two instructions in the instruction cache, the PDU uses two bits to record the state of the branch prediction algorithm. These bits are stored in the cache, in a tag associated with the instructions.

For each four instructions in the instruction cache, a tag field is provided called Next Address. The PDU computes the target address of a branch instruction when the instruction is first fetched for execution, and it records this address in the Next Address field. This field makes it possible to continue prefetching instructions in subsequent passes, without having to recompute the target address each time. Since there is only one Next Address field for each half of a cache line, its benefit can be fully realized only if there is at most one branch instruction in each group of four instructions.

Grouping

In the third stage of the pipeline, stage G, the Grouping Logic selects a group of up to four instructions to be executed in parallel and dispatches them to the integer and floating-point execution units. Figure 8.25 shows a short instruction sequence and the way these instructions would be dispatched. Parts *b* and *c* of the figure show the instruction grouping when the PDU predicts that the branch will be taken and not taken, respectively. Note that the instruction in the delay slot, FCMP, is included in the selected group in both cases. It will be executed, but not committed until the branch decision is made. Its results will be annulled if the branch is not taken, because the Annul bit in the branch instruction is set to 1. The first two instructions in each group are dispatched to the integer unit and the next two to the floating-point unit.

```
        ADDcc    R3, R4, R7   R7 ← [R3] + [R4],
                              Set condition codes
        BRZ,a    Label        Branch if zero, set Annul bit to 1
        FCMP     F1, F5       FP: Compare [F2] and [F5]
        FADD     F2, F3, F6   FP: F6 ← [F2] + [F3]
        FMOVs    F3, F4       Move single precision operand from F3 to F4
          ⋮
Label   FSUB     F2, F3, F6   FP: F6 ← [F2] − [F3]
        LDSW     R3, R4, R7   Load single word at location [R3] + [R4] into R7
          ⋮
```

(a) Program fragment

```
        ADDcc    R3, R4, R7
        BRZ,a    Label
        FCMP     F1, F5
        FSUB     F2, F3, F6
```

(b) Instruction grouping, branch taken

```
        ADDcc    R3, R4, R7
        BRZ,a    Label
        FCMP     R1, R5
        FADD     R2, R3, R6
```

(c) Instruction grouping, branch not taken

Figure 8.25 Example of instruction grouping.

The grouping logic circuit is responsible for ensuring that the instructions it dispatches are ready for execution. For example, all the operands referenced by the instruction in a group must be available. No two instructions can be included in the same group if one of them depends on the result of the other. Branch instructions are excepted from this condition, as will be explained shortly.

Instructions are dispatched in program order. Recall that if a group includes a branch instruction, that instruction will have already been tentatively executed as a result of branch prediction in the prefetch and decode unit. Hence, the instructions in the instruction buffer will be in correct order based on this prediction. The grouping logic simply examines the instructions in the instruction buffer in order, with the objective of selecting the largest number at the head of the queue that satisfy the grouping constraints.

Some of the constraints that the grouping logic takes into account in selecting instructions to include in a group are:

1. Instructions can only be dispatched in sequence. If one instruction cannot be included in a group, no later instruction can be selected.

2. The source operand of an instruction cannot depend on the destination operand of any other instruction in the same group. There are two exceptions to this rule:

- A store instruction, which stores the contents of a register in the memory, may be grouped with an earlier instruction that has that register as a destination. This is allowed because, as we will see shortly, the store instruction does not require the data until a later stage in the pipeline.
- A branch instruction may be grouped with an earlier instruction that sets the condition codes.

3. No two instructions in a group can have the same destination operand, unless the destination is register R0. For example, the LDSW instruction in Figure 8.26*a* cannot be grouped with the ADD instruction and must be delayed to the next group as shown.

4. In some cases, certain instructions must be delayed two or three clock cycles relative to other instructions. For example, the conditional instruction

MOVRZ R1, R6, R7

(Move on register condition) moves the contents of R6 into R7 if the contents of R1 are equal to zero. This instruction requires an additional clock cycle to check if the contents

ADD	R3, R5, R6	G	E	C	N1	N2	N3	W	
LDSW	R4, R7, R6		G	E	C	N1	N2	N3	W

(a) Instructions with common destination

MOVRZ	R1, R6, R7	G	E	C	N1	N2	N3	W		
OR	R7, R8, R9			G	E	C	N1	N2	N3	W

(b) Delay caused by MOVR instruction

Figure 8.26 Dispatch delays due to hazards.

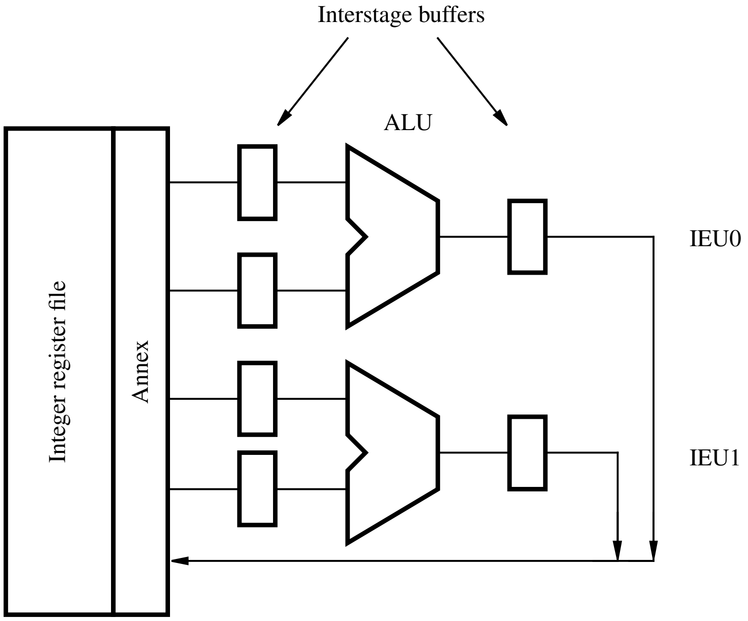


Figure 8.27 Integer execution unit.

of R1 are equal to zero. Hence, an instruction that reads register R7 cannot be in the same group or in the following group. The earliest dispatch for such an instruction is as shown in Figure 8.26*b*.

When the grouping logic dispatches an instruction to the integer unit, it also fetches the source operands of that instruction from the integer register file. The information needed to access the register file is available in the decoded bits that were entered into the instruction buffer by the prefetch and decode unit. Thus, by the end of the clock cycle of stage G, one or two integer instructions will be ready to enter the execution phase. The data read from the register file are stored in interstage buffers, as shown in Figure 8.27. Access to operands in the floating-point register file takes place in stage R, after the instruction has been forwarded to the floating-point unit.

Execution Units

The Integer execution unit consists of two similar but not identical units, IEU0 and IEU1. Only unit IEU0 is equipped to handle shift instructions, while only IEU1 can generate condition codes. Instructions that do not involve these operations can be executed in either unit.

The ALU operation for most integer instructions is completed in one clock cycle. This is stage E in the pipeline. At the end of this clock cycle, the result is stored in the buffer shown at the output of the ALU in Figure 8.27. In the next clock cycle, stage C, the contents of this buffer are transferred to a part of the register file called the Annex. The Annex contains the temporary registers used in register renaming, as explained in

Section 8.6. The contents of a temporary register are transferred to the corresponding permanent register in stage W of the pipeline.

Another action that takes place during stage C is the generation of condition codes. Of course, this is done only for instructions such as ADDcc, which specify that the condition code flags are to be set. Such instructions must be executed in unit IEU1.

Consider an instruction Icc that sets the condition code flags and a subsequent conditional branch instruction, BRcc, that checks these flags. When BRcc is encountered by the prefetch and dispatch unit, the results of execution of Icc may not yet be available. The PDU predicts the outcome of the branch and continues prefetching instructions on that basis. Later, the condition codes are generated when Icc reaches stage C of the pipeline, and they are sent to the PDU during the same clock cycle. The PDU checks whether its branch prediction was correct. If it was, execution continues without interruption. Otherwise, the contents of the pipeline and the instruction buffer are flushed, and the PDU begins to fetch the correct instructions. Aborting instructions at this point is possible because these instructions will not have reached stage W of the pipeline.

When a branch is incorrectly predicted, many instructions may be incorrectly prefetched and partially executed. The situation is illustrated in Figure 8.28. We have assumed that the grouping logic has been able to dispatch four instructions in three successive clock cycles. Instruction Icc at the beginning of the first group sets the condition codes, which are tested by the following instruction, BRcc. The test is performed when the first group reaches stage C. At this time, the third group, I_9 to I_{12}, is entering stage G of the pipeline. If the branch prediction was incorrect, the nine instructions I_4 to I_{12} will be aborted (recall that instruction I_3 in the delay slot is always executed). In addition, any instructions that may have been prefetched and loaded into the instruction buffer will also be discarded. Hence, in the extreme case, up to 21 intrustions may be discarded.

No operation is performed in pipeline stages N1 and N2. These stages introduce a delay of two clock cycles, to make the total length of the integer pipeline the same

I_1 (Icc)	G	E	C
I_2 (BRcc)	G	E	C
I_3	G	E	C
I_4	G	E	C
I_5		G	E
I_6		G	E
I_7		G	E
I_8		G	E
I_9			G
I_{10}			G
I_{11}			G
I_{12}			G

↑ Abort

Figure 8.28 Worst-case timing for an incorrectly predicted branch.

as that of the floating-point pipeline. For integer instructions that do not complete their execution in stage C, such as divide instructions, execution continues through stages N1 and N2. If more time is needed, additional clock cycles are inserted between N1 and N2. The instruction enters N2 only in the last clock cycle of its execution. For example, if the operation performed by an instruction requires 16 clock cycles, 12 clock cycles are inserted after stage N1.

The Floating-point execution unit also has two independent pipelines. Register operands are fetched in stage R, and the operation is performed in up to three pipeline stages (X1 to X3). Here also, if additional clock cycles are needed, such as for the square-root instruction, additional clock cycles are inserted between X2 and X3.

In stage N3, the processor examines various exception conditions to determine whether a trap (interrupt) should be taken. Finally, the result of an instruction is stored in the destination location, either in a register or in the data cache, during the Write stage (W). An instruction may be aborted and all its effects annulled at any time up to this stage. Once the Write stage is entered, the execution of the instruction cannot be stopped.

Load and Store Unit

The instruction

LDUW R5, R6, R7

loads an unsigned 32-bit word from location [R5] + [R6] in the memory into register R7. As for other integer instructions, the contents of registers R5 and R6 are fetched during stage G of the pipeline. However, instead of this data being sent to one of the integer execution units, the instruction and its operands are forwarded to the Load and Store Unit, shown in Figure 8.29. The unit begins by adding the contents of registers R5 and R6 during stage E to generate the effective address of the memory location to be accessed. The result is a virtual address value, which is sent to the data cache. At the same time, it is sent to the data lookaside buffer, dTLB, to be translated into a physical address.

Data are stored in the cache according to their virtual address, so that they can be accessed quickly without waiting for address translation to be completed. Both the data and the corresponding tag information are read from the D-cache in stage C, and the physical address is read from the dTLB. The tag used in the D-cache is a part of the physical address of the data. During stage N1, the tag read from the D-cache is checked against the physical address obtained from the dTLB. In the case of a hit, the data are loaded into an Annex register, to be transferred to the destination register in stage W. If the tags do not match, the instruction enters the Load/store queue, where it waits for a cache block to be loaded from the external cache into the D-cache.

Once an instruction enters the Load/store queue it is no longer considered to be in the execution pipeline. Other instructions may proceed to completion while a load instruction is waiting in the queue, unless one of these instructions references the register awaiting data from the memory (R7 in the example above). Thus, the Load/store queue decouples the operation of the pipeline from external data access operations so that the two can proceed independently.

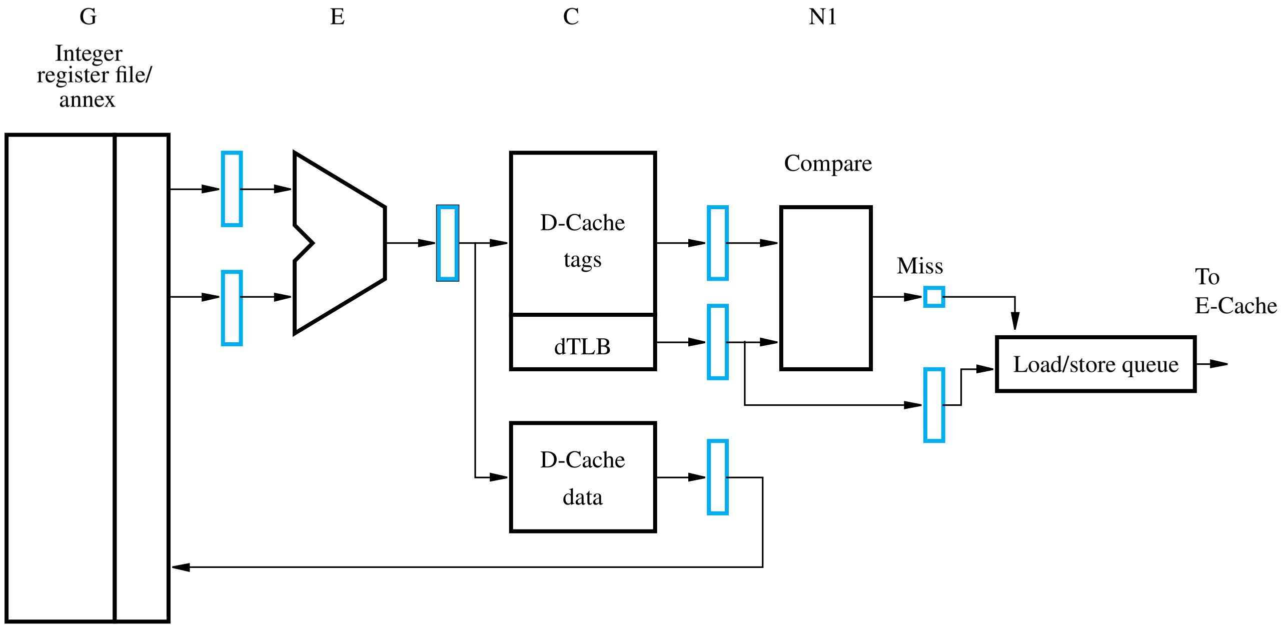


Figure 8.29 Load and store unit.

Execution Flow

It is instructive to examine the flow of instructions and data in the UltraSPARC II processor and between it and the external cache and the memory. Figure 8.30 shows the main functional units of Figure 8.23 reorganized to illustrate the flow of instructions and data and the role that the instruction and data queues play.

Instructions are fetched from the I-cache and loaded into the instruction buffer, which can store up to 12 instructions. From there, instructions are forwarded, up to four at a time, to the block labeled "Internal registers and execution units," where they are executed. On average, the speed with which the PDU can fill the instruction buffer is higher than the speed with which the grouping logic dispatches instructions. Hence, the instruction buffer tends to be full most of the time. In the absence of cache misses and mispredicted branches, the internal execution units are never starved for instructions. Similarly, the memory operands of load and store instructions are likely to be found in the data cache most of the time, where they are accessed in one clock cycle. Hence execution proceeds without delay.

When a miss occurs in the instruction cache, there is a delay of a few clock cycles while the appropriate block is loaded from the external cache. During that time, the

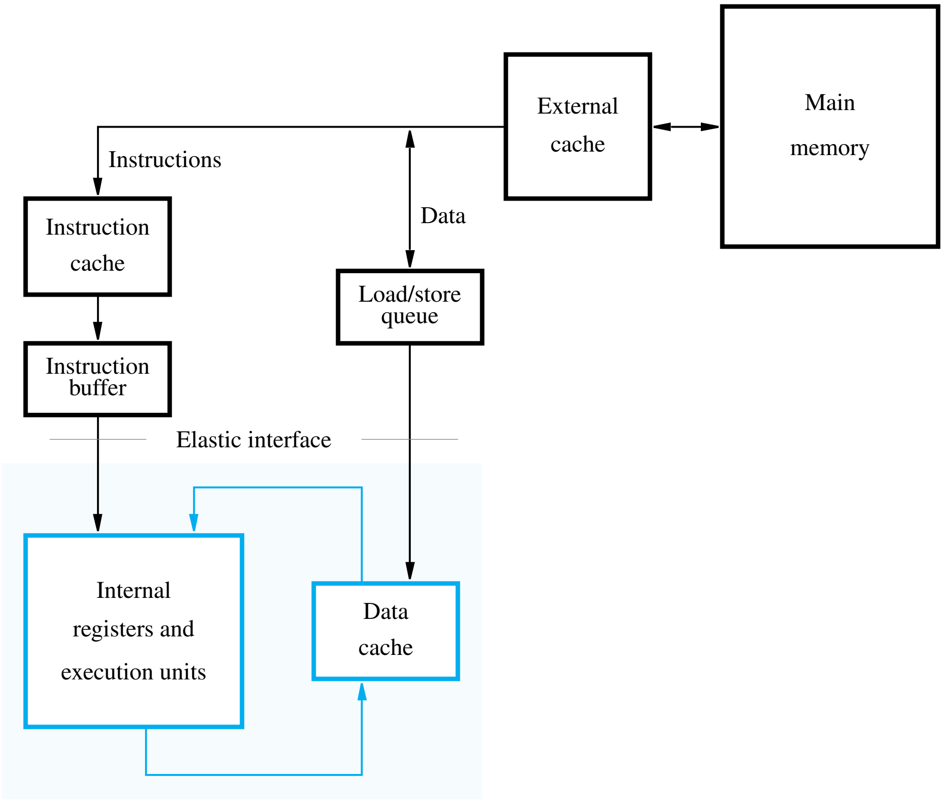


Figure 8.30 Execution flow.

grouping logic continues to dispatch instructions from the instruction buffer until the buffer becomes empty. It takes three or four clock cycles to load a cache block (eight instructions) from the external cache, depending on the processor model. This is about the same length of time it takes the grouping logic to dispatch the instructions in a full instruction buffer. (Recall that it is not always possible to dispatch four instructions in every clock cycle.) Hence, if the instruction buffer is full at the time a cache miss occurs, operation of the execution pipeline may not be interrupted at all. If a miss also occurs in the external cache, considerably more time will be needed to access the memory. In this case, it is inevitable that the pipeline will be stalled.

A load operation that causes a cache miss enters the Load/store queue and waits for a transfer from the external cache or the memory. However, as long as the destination register of the load operation is not referenced by later instructions, internal instruction execution continues. Thus, the instruction buffer and the Load/store queue isolate the internal processor pipeline from external data transfers. They act as elastic interfaces that allow the internal high-speed pipeline to continue to run while slow external data transfers are taking place.

8.8 PERFORMANCE CONSIDERATIONS

We pointed out in Section 1.6 that the execution time, T, of a program that has a dynamic instruction count N is given by

$$T = \frac{N \times S}{R}$$

where S is the average number of clock cycles it takes to fetch and execute one instruction, and R is the clock rate. This simple model assumes that instructions are executed one after the other, with no overlap. A useful performance indicator is the *instruction throughput,* which is the number of instructions executed per second. For sequential execution, the throughput, P_s is given by

$$P_s = R/S$$

In this section, we examine the extent to which pipelining increases instruction throughput. However, we should reemphasize the point made in Chapter 1 regarding performance measures. The only real measure of performance is the total execution time of a program. Higher instruction throughput will not necessarily lead to higher performance if a larger number of instructions is needed to implement the desired task. For this reason, the SPEC ratings described in Chapter 1 provide a much better indicator when comparing two processors.

Figure 8.2 shows that a four-stage pipeline may increase instruction throughput by a factor of four. In general, an n-stage pipeline has the potential to increase throughput n times. Thus, it would appear that the higher the value of n, the larger the performance gain. This leads to two questions:

- How much of this potential increase in instruction throughput can be realized in practice?
- What is a good value for n?

Any time a pipeline is stalled, the instruction throughput is reduced. Hence, the performance of a pipeline is highly influenced by factors such as branch and cache miss penalties. First, we discuss the effect of these factors on performance, and then we return to the question of how many pipeline stages should be used.

8.8.1 Effect of Instruction Hazards

The effects of various hazards have been examined qualitatively in the previous sections. We now assess the impact of cache misses and branch penalties in quantitative terms.

Consider a processor that uses the four-stage pipeline of Figure 8.2. The clock rate, hence the time allocated to each step in the pipeline, is determined by the longest step. Let the delay through the ALU be the critical parameter. This is the time needed to add two integers. Thus, if the ALU delay is 2 ns, a clock of 500 MHz can be used. The on-chip instruction and data caches for this processor should also be designed to have an access time of 2 ns. Under ideal conditions, this pipelined processor will have an instruction throughput, P_p, given by

$$P_p = R = 500 \text{ MIPS (million instructions per second)}$$

To evaluate the effect of cache misses, we use the same parameters as in Section 5.6.2. The cache miss penalty, M_p, in that system is computed to be 17 clock cycles. Let T_I be the time between two successive instruction completions. For sequential execution, $T_I = S$. However, in the absence of hazards, a pipelined processor completes the execution of one instruction each clock cycle, thus, $T_I = 1$ cycle. A cache miss stalls the pipeline by an amount equal to the cache miss penalty. This means that the value of T_I increases by an amount equal to the cache miss penalty for the instruction in which the miss occurs. A cache miss can occur for either instructions or data. Consider a computer that has a shared cache for both instructions and data, and let d be the percentage of instructions that refer to data operands in the memory. The average increase in the value of T_I as a result of cache misses is given by

$$\delta_{miss} = ((1 - h_i) + d(1 - h_d)) \times M_p$$

where h_i and h_d are the hit ratios for instructions and data, respectively. Assume that 30 percent of the instructions access data in memory. With a 95-percent instruction hit rate and a 90-percent data hit rate, δ_{miss} is given by

$$\delta_{miss} = (0.05 + 0.3 \times 0.1) \times 17 = 1.36 \text{ cycles}$$

Taking this delay into account, the processor's throughput would be

$$P_p = \frac{R}{T_I} = \frac{R}{1 + \delta_{miss}} = 0.42R$$

Note that with R expressed in MHz, the throughput is obtained directly in millions of instructions per second. For $R = 500$ MHz, $P_p = 210$ MIPS.

Let us compare this value to the throughput obtainable without pipelining. A processor that uses sequential execution requires four cycles per instruction. Its throughput would be

$$P_s = \frac{R}{4 + \delta_{miss}} = 0.19R$$

For $R = 500$ MHz, $P_s = 95$ MIPS. Clearly, pipelining leads to significantly higher throughput. But the performance gain of $0.42/0.19 = 2.2$ is only slightly better than one-half the ideal case.

Reducing the cache miss penalty is particularly worthwhile in a pipelined processor. As Chapter 5 explains, this can be achieved by introducing a secondary cache between the primary, on-chip cache and the memory. Assume that the time needed to transfer an 8-word block from the secondary cache is 10 ns. Hence, a miss in the primary cache for which the required block is found in the secondary cache introduces a penalty, M_s, of 5 cycles. In the case of a miss in the secondary cache, the full 17-cycle penalty (M_p) is still incurred. Hence, assuming a hit rate h_s of 94 percent in the secondary cache, the average increase in T_I is

$$\delta_{miss} = ((1 - h_i) + d(1 - h_d)) \times (h_s \times M_s + (1 - h_s) \times M_p) = 0.46 \text{ cycle}$$

The instruction throughput in this case is $0.68R$, or 340 MIPS. An equivalent nonpipelined processor would have a throughput of $0.22R$, or 110 MIPS. Thus, pipelining provides a performance gain of $0.68/0.22 = 3.1$.

The values of 1.36 and 0.46 are, in fact, somewhat pessimistic, because we have assumed that every time a data miss occurs, the entire miss penalty is incurred. This is the case only if the instruction immediately following the instruction that references memory is delayed while the processor waits for the memory access to be completed. However, an optimizing compiler attempts to increase the distance between two instructions that create a dependency by placing other instructions between them whenever possible. Also, in a processor that uses an instruction queue, the cache miss penalty during instruction fetches may have a much reduced effect as the processor is able to dispatch instructions from the queue.

8.8.2 NUMBER OF PIPELINE STAGES

The fact that an n-stage pipeline may increase instruction throughput by a factor of n suggests that we should use a large number of stages. However, as the number of pipeline stages increases, so does the probability of the pipeline being stalled, because more instructions are being executed concurrently. Thus, dependencies between instructions that are far apart may still cause the pipeline to stall. Also, branch penalties may become more significant, as Figure 8.9 shows. For these reasons, the gain from increasing the value of n begins to diminish, and the associated cost is not justified.

Another important factor is the inherent delay in the basic operations performed by the processor. The most important among these is the ALU delay. In many processors, the cycle time of the processor clock is chosen such that one ALU operation can be completed in one cycle. Other operations are divided into steps that take about the same time as an add operation. It is also possible to use a pipelined ALU. For example, the ALU of the Compaq Alpha 21064 processor consists of a two-stage pipeline, in which each stage completes its operation in 5 ns.

Many pipelined processors use four to six stages. Others divide instruction execution into smaller steps and use more pipeline stages and a faster clock. For example, the UltraSPARC II uses a 9-stage pipeline and Intel's Pentium Pro uses a 12-stage pipeline. The latest Intel processor, Pentium 4, has a 20-stage pipeline and uses a clock speed in the range 1.3 to 1.5 GHz. For fast operations, there are two pipeline stages in one clock cycle.

8.9 CONCLUDING REMARKS

Two important features have been introduced in this chapter, pipelining and multiple issue. Pipelining enables us to build processors with instruction throughput approaching one instruction per clock cycle. Multiple issue makes possible superscalar operation, with instruction throughput of several instructions per clock cycle.

The potential gain in performance can only be realized by careful attention to three aspects:

- The instruction set of the processor
- The design of the pipeline hardware
- The design of the associated compiler

It is important to appreciate that there are strong interactions among all three. High performance is critically dependent on the extent to which these interactions are taken into account in the design of a processor. Instruction sets that are particularly well-suited for pipelined execution are key features of modern processors.

PROBLEMS

8.1 Consider the following sequence of instructions

```
Add   #20,R0,R1
Mul   #3,R2,R3
And   #$3A,R2,R4
Add   R0,R2,R5
```

In all instructions, the destination operand is given last. Initially, registers R0 and R2 contain 2000 and 50, respectively. These instructions are executed in a computer that has a four-stage pipeline similar to that shown in Figure 8.2. Assume that the first instruction is fetched in clock cycle 1, and that instruction fetch requires only one clock cycle.

(*a*) Draw a diagram similar to Figure 8.2*a*. Describe the operation being performed by each pipeline stage during each of clock cycles 1 through 4.

(*b*) Give the contents of the interstage buffers, B1, B2, and B3, during clock cycles 2 to 5.

8.2 Repeat Problem 8.1 for the following program:

```
Add   #20,R0,R1
Mul   #3,R2,R3
And   #$3A,R1,R4
Add   R0,R2,R5
```

8.3 Instruction I_2 in Figure 8.6 is delayed because it depends on the results of I_1. By occupying the Decode stage, instruction I_2 blocks I_3, which, in turn, blocks I_4. Assuming that I_3 and I_4 do not depend on either I_1 or I_2 and that the register file allows two Write steps to proceed in parallel, how would you use additional storage buffers to make it possible for I_3 and I_4 to proceed earlier than in Figure 8.6? Redraw the figure, showing the new order of steps.

8.4 The delay bubble in Figure 8.6 arises because instruction I_2 is delayed in the Decode stage. As a result, instructions I_3 and I_4 are delayed even if they do not depend on either I_1 or I_2. Assume that the Decode stage allows two Decode steps to proceed in parallel. Show that the delay bubble can be completely eliminated if the register file also allows two Write steps to proceed in parallel.

8.5 Figure 8.4 shows an instruction being delayed as a result of a cache miss. Redraw this figure for the hardware organization of Figure 8.10. Assume that the instruction queue can hold up to four instructions and that the instruction fetch unit reads two instructions at a time from the cache.

8.6 A program loop ends with a conditional branch to the beginning of the loop. How would you implement this loop on a pipelined computer that uses delayed branching with one delay slot? Under what conditions would you be able to put a useful instruction in the delay slot?

8.7 The branch instruction of the UltraSPARC II processor has an Annul bit. When set by the compiler, the instruction in the delay slot is discarded if the branch is not taken. An alternative choice is to have the instruction discarded if the branch is taken. When is each of these choices advantageous?

8.8 A computer has one delay slot. The instruction in this slot is always executed, but only on a speculative basis. If a branch does not take place, the results of that instruction are discarded. Suggest a way to implement program loops efficiently on this computer.

8.9 Rewrite the sort routine shown in Figure 2.34 for the SPARC processor. Recall that the SPARC architecture has one delay slot with an associated Annul bit and uses branch prediction. Attempt to fill the delay slots with useful instructions wherever possible.

8.10 Consider a statement of the form

IF A>B THEN action 1 ELSE action 2

Write a sequence of assembly language instructions, first using branch instructions only, then using conditional instructions such as those available on the ARM processor.

Assume a simple two-stage pipeline, and draw a diagram similar to that in Figure 8.8 to compare execution times for the two approaches.

8.11 The feed-forward path in Figure 8.7 (blue lines) allows the content of the RSLT register to be used directly in an ALU operation. The result of that operation is stored back in the RSLT register, replacing its previous contents. What type of register is needed to make such an operation possible?

Consider the two instructions

I_1:	Add	R1,R2,R3
I_2:	Shift_left	R3

Assume that before instruction I_1 is executed, R1, R2, R3, and RSLT contain the values 30, 100, 45, and 198, respectively. Draw a timing diagram for a 4-stage pipeline, showing the clock signal and the contents of the RSLT register during each cycle. Use your diagram to show that correct results will be obtained during the forwarding operation.

8.12 Write the program in Figure 2.37 for a processor in which only load and store instructions access memory. Identify all dependencies in the program and show how you would optimize it for execution on a pipelined processor.

8.13 Assume that 20 percent of the dynamic count of the instructions executed on a computer are branch instructions. Delayed branching is used, with one delay slot. Estimate the gain in performance if the compiler is able to use 85 percent of the delay slots.

8.14 A pipelined processor has two branch delay slots. An optimizing compiler can fill one of these slots 85 percent of the time and can fill the second slot only 20 percent of the time. What is the percentage improvement in performance achieved by this optimization, assuming that 20 percent of the instructions executed are branch instructions?

8.15 A pipelined processor uses the delayed branch technique. You are asked to recommend one of two possibilities for the design of this processor. In the first possibility, the processor has a 4-stage pipeline and one delay slot, and in the second possibility, it has a 6-stage pipeline with two delay slots. Compare the performance of these two alternatives, taking only the branch penalty into account. Assume that 20 percent of the instructions are branch instructions and that an optimizing compiler has an 80 percent success rate in filling the single delay slot. For the second alternative, the compiler is able to fill the second slot 25 percent of the time.

8.16 Consider a processor that uses the branch prediction mechanism represented in Figure 8.15*b*. The initial state is either LT or LNT, depending on information provided in the branch instruction. Discuss how the compiler should handle the branch instructions used to control "do while" and "do until" loops, and discuss the suitability of the branch prediction mechanism in each case.

8.17 Assume that the instruction queue in Figure 8.10 can hold up to six instructions. Redraw Figure 8.11 assuming that the queue is full in clock cycle 1 and that the fetch unit can read up to two instructions at a time from the cache. When will the queue become full again after instruction I_k is fetched?

8.18 Redraw Figure 8.11 for the case of the mispredicted branch in Figure 8.14.

8.19 Figure 8.16 shows that one instruction that uses a complex addressing mode takes the same time to execute as an equivalent sequence of instructions that use simpler addressing modes. Yet, the use of simple addressing modes is one of the tenets of the RISC philosophy. How would you design a pipeline to handle complex addressing modes? Discuss the pros and cons of this approach.

REFERENCE

1. The SPARC Architecture Manual, Version 9, D. Weaver and T. Germond, ed., PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994.

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, ..., 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 is obtained by modifying the program in Figure 9.11 as follows:

```
/* Define register addresses */
#define RBUF (volatile char *) 0xFFFFFFE0
#define SSTAT (volatile char *) 0xFFFFFFE2
#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 *) 0xFFFFFFE0
#define SCONT (char *) 0xFFFFFFE3
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#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 */
    *SCONT = 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 *) 0xFFFFFFE0
#define SSTAT (volatile char *) 0xFFFFFFE2
#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 *) 0xFFFFFFE0
#define SCONT (char *) 0xFFFFFFE3
#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 */
    *SCONT = 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 *) 0xFFFFFFE0
#define SSTAT (volatile char *) 0xFFFFFFE2
#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    #$48,R0        Check if H.
          Branch≠0   LOOP
LOOP2     Testbit    #0,SSTAT       Check if first digit is ready.
          Branch=0   LOOP2
          MoveByte   RBUF,R0        Read the first digit.
          LShiftL    #4,R0          Shift left 4 bit positions.
LOOP3     Testbit    #0,SSTAT       Check if second digit is ready.
          Branch=0   LOOP3
          MoveByte   RBUF,R1        Read the second digit.
          And        #$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 *) 0xFFFFFFE0
#define SCONT (char *) 0xFFFFFFE3
#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];                    /* 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 */
    *SCONT = 0x10;                     /* Enable receiver interrupts */

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

```
/* 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, recpectively. Then, the required task can be achieved by modifying the program in Figure 9.11 as follows:

```
/* Define register addresses */
#define RBUF (volatile char *) 0xFFFFFFE0
#define SSTAT (volatile char *) 0xFFFFFFE2
#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];                           /* 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 */

    /* 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 */
            j = *RBUF & 0xF;                  /* Extract the BCD value */
            digits[i] = seg7[j];              /* Save 7-segment code for the digit */
          }
          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 */
          }
        }
    }
}
```

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, recpectively. 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 digi ts
         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 d3.
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, recpectively. 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 *) 0xFFFFFFE0
#define SCONT (char *) 0xFFFFFFE3
#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 */
    *SCONT = 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, recpectively. 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 d3.
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 *) 0xFFFFFFE0
#define SSTAT (volatile char *) 0xFFFFFFE2
#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≠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.
```