# Chapter 6 – Arithmetic

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

| | | | | | |
|---|---|---|---|---|---|
| 010110 | (+22) | 101011 | (−21) | 111111 | (−1) |
| + 001001 | + (+9) | + 100101 | + (−27) | + 000111 | + (+7) |
| 011111 | (+31) | 010000 | (−48) | 000110 | (+6) |
| | | overflow | | | |

| | | | | | |
|---|---|---|---|---|---|
| 011001 | (+25) | 110111 | (−9) | 010101 | (+21) |
| + 010000 | + (+16) | + 111001 | + (−7) | + 101011 | + (−21) |
| 101001 | (+41) | 110000 | (−16) | 000000 | (0) |
| overflow | | | | | |

| | | |
|---|---|---|
| 010110 | (+22) | 010110 |
| − 011111 | − (+31) | + 100001 |
| | (−9) | 110111 |

| | | |
|---|---|---|
| 111110 | (−2) | 111110 |
| − 100101 | − (−27) | + 011011 |
| | (+25) | 011001 |

| | | |
|---|---|---|
| 100001 | (−31) | 100001 |
| − 011101 | − (+29) | + 100011 |
| | (−60) | 000100 |
| | | overflow |

| | | |
|---|---|---|
| 111111 | (−1) | 111111 |
| − 000111 | − (+7) | + 111001 |
| | (−8) | 111000 |

| | | |
|---|---|---|
| 000111 | (+7) | 000111 |
| − 111000 | − (−8) | + 001000 |
| | (+15) | 001111 |

| | | |
|---|---|---|
| 011010 | (+26) | 011010 |
| − 100010 | − (−30) | + 011110 |
| | (+56) | 111000 |
| | | overflow |

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 |

2

6.4. Ternary numbers with addition and subtraction operations:

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

Addition operations:

```
  002002        002002        002002
+ 221122      + 011112      + 211110
---------     ---------     ---------
  000201        020121        220112


  221122        221122        011112
+ 011112      + 211110      + 211110
---------     ---------     ---------
  010011        210002        222222
```

Subtraction operations:

```
  002002        002002
− 221122      + 001101
---------     ---------
                010110


  002002        002002
− 011112      + 211111
---------     ---------
                220120


  002002        002002
− 211110      + 011120
---------     ---------
                020122


  221122        221122
− 011112      + 211111
---------     ---------
                210010


  221122        221122
− 211110      + 011120
---------     ---------
                010012


  011112        011112
− 211110      + 011120
---------     ---------
                100002
              overflow
```

6.5. (*a*)

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

$$s = x \oplus y$$
$$c = x\,y$$
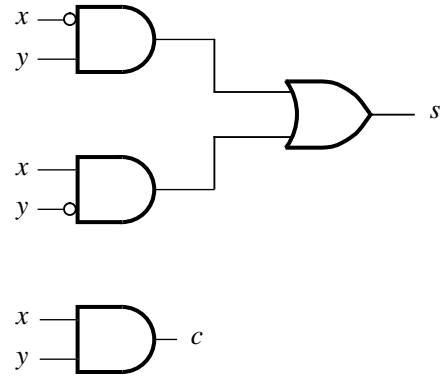


(*b*)
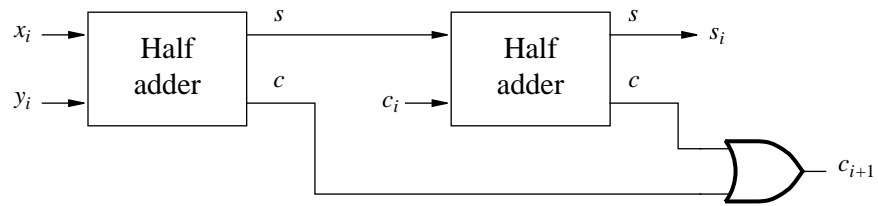


(*c*) The longest path through the circuit in Part (*b*) is 6 gate delays (including input inversions) in producing $s_i$; and the longest path through the circuit in Figure 6.2*a* is 3 gate delays in producing $s_i$, assuming that $s_i$ is implemented as a two-level AND-OR circuit, and including input 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] = $D_0$. |
| | MOV | R2,#&F | Load mask bits into R2. |
| | MOV | R4,#10 | Load multipliers |
| | MOV | R5,#100 | into R4, R5, and R6. |
| | MOV | R6,#1000 | |
| | AND | R3,R2,R1,LSR #4 | Get $D_1$ into R3. |
| | MLA | R0,R3,R4,R0 | Add $10D_1$ into R0. |
| | AND | R3,R2,R1,LSR #8 | Get $D_2$ into R3. |
| | MLA | R0,R3,R5,R0 | Add $100D_2$ into R0. |
| | AND | R3,R2,R1,LSR #12 | Get $D_3$ into R3. |
| | MLA | R0,R3,R6,R0 | Add $1000D_3$ into 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] = D_0.
           SHR    ESI,4
           MOV    EAX,ESI
           AND    EAX,FH
           MUL    [EBP − 4]
           ADD    EDX,EAX          [EDX] = binary of D_1 D_0.
           SHR    ESI,4
           MOV    EAX,ESI
           AND    EAX,FH
           MUL    [EBP − 8]
           ADD    EDX,EAX          [EDX] = binary of D_2 D_1 D_0.
           SHR    ESI,4
           MOV    EAX,ESI
           AND    EAX,FH
           MUL    [EBP − 12]
           ADD    EDX,EAX          [EDX] = binary of D_3 D_2 D_1 D_0.
           MOV    EAX,[EBP + 12]   Store converted
           MOV    [EAX],EDX         value into BINARY.
           POP    EAX              Restore registers.
           POP    ESI
           POP    EDX
           ADD    ESP,12           Remove local parameters.
           POP    EBP              Restore EBP.
           RET                     Return.
```

AND EDX,FH → $[EDX] = D_0$.

ADD EDX,EAX → $[EDX] = $ binary of $D_1 D_0$.

ADD EDX,EAX → $[EDX] = $ binary of $D_2 D_1 D_0$.

ADD EDX,EAX → $[EDX] = $ binary of $D_3 D_2 D_1 D_0$.

($iii$) The 68000 subroutine uses a loop structure to convert the four-digit decimal integer $D_3 D_2 D_1 D_0$ (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 $6_{10}$.
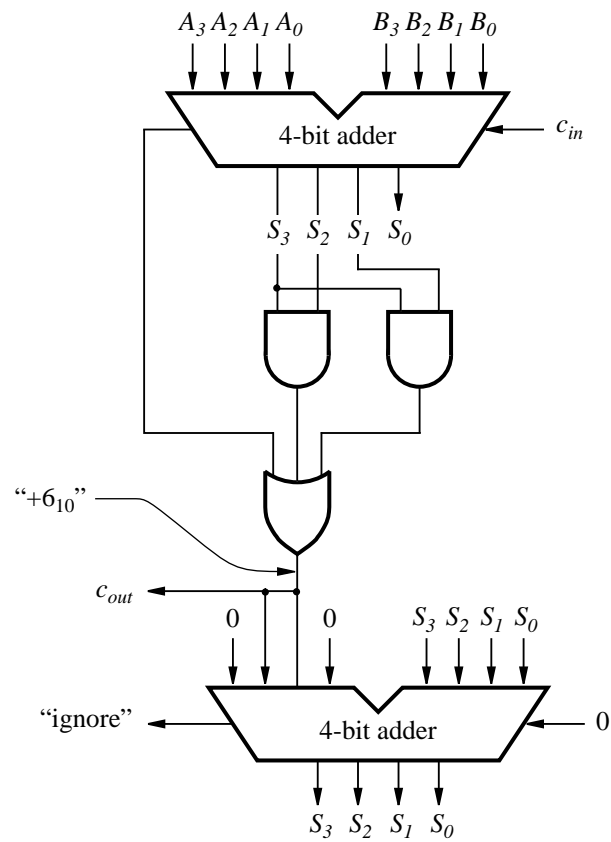
(*b*)

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

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

output carry = 1

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

(*c*)



9

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:

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

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

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

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

6.11. (a) Each B cell requires 3 gates as shown in Figure 6.4a. The carries $c_1$, $c_2$, $c_3$, and $c_4$, require 2, 3, 4, and 5, gates, respectively; and the outputs $G_0^I$ and $P_0^I$ require 4 and 1 gates, as seen from the logic expressions in Section 6.2.1. Therefore, a total of $12 + 19 = 31$ gates are required for the 4-bit adder.

(b) Four 4-bit adders require $4 \times 31 = 124$ gates, and the carry-lookahead logic block requires 19 gates because it has the same structure as the lookahead block in Figure 6.4. Total gate count is thus 143. However, we should subtract $4 \times 5 = 20$ gates from this total corresponding to the logic for $c_4$, $c_8$, $c_{12}$, and $c_{16}$, that is in the 4-bit adders but which is replaced by the lookahead logic in Figure 6.5. Therefore, total gate count for the 16-bit adder is $143 - 20 = 123$ gates.

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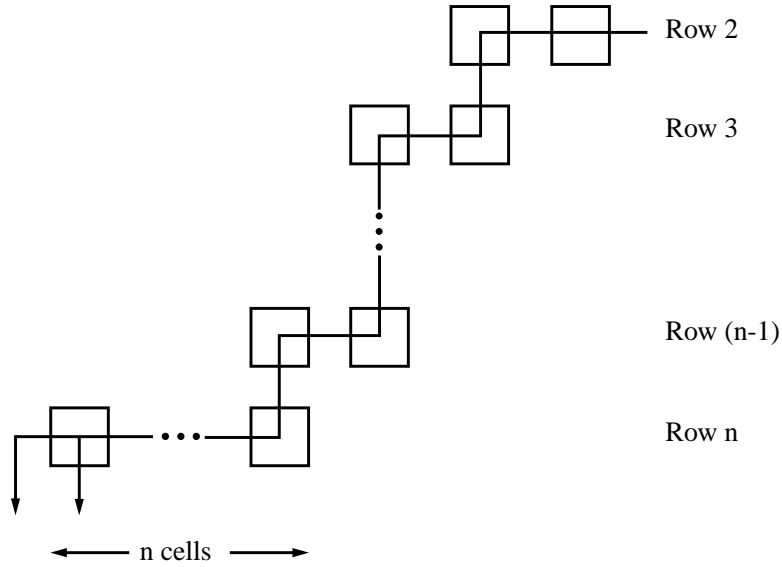


Each of the two FA blocks in rows 2 through $n - 1$ introduces 2 gate delays, for a total of $4(n - 2)$ gate delays. Row $n$ introduces $2n$ gate delays. Adding in the initial AND gate delay for row 1 and all other cells, total delay is:

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

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

$$
\begin{array}{rr}
\text{B} \;=\; 00101 & (\,5) \\
\underline{\times\text{A} \;=\; \underline{10101}} & \underline{\times\;(21)} \\
00101 & (105) \\
0 & \\
00101\phantom{00} & \\
\underline{001010\phantom{000}} & \\
001101001 & (105)
\end{array}
$$

$$
\begin{array}{r}
100 \\
00101\,\overline{)10101} \\
\underline{101}\phantom{00} \\
00001
\end{array}
\qquad
\begin{array}{r}
4 \\
5\,\overline{)21} \\
\underline{20} \\
1
\end{array}
$$

6.14. The multiplication and division charts are:

$A \times B$ :

|   | M |   |   |
|---|---|---|---|
|   | 00101 |   |   |
| 0 | 00000 | 10101 | Initial configuration |
| C | A | Q |   |

| 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 | |
|---|---|---|---|
| | 000000 | 10101 | |

Initial configuration

|   | 000101 |   |
|---|--------|---|
|   | M |   |

| | A | Q | |
|---|---|---|---|
| shift | 000001 | 0 1 0 1 □ | |
| subtract | 111011 | | 1st cycle |
| | 111100 | 0 1 0 1 0 | |
| shift | 111000 | 1 0 1 0 □ | |
| add | 000101 | | 2nd cycle |
| | 111101 | 1 0 1 0 0 | |
| shift | 111011 | 0 1 0 0 □ | |
| **add** | **000101** | | 3rd cycle |
| | **000000** | 0 1 0 0 1 | |
| shift | 000000 | 1 0 0 1 □ | |
| subtract | 111011 | | 4th cycle |
| | 111011 | 1 0 0 1 0 | |
| shift | 110111 | 0 0 1 0 □ | |
| add | 000101 | | 5th cycle |
| | 111100 | 0 0 1 0 0 | |
| **add** | **000101** | | |
| | 000001 | quotient | |

remainder

13

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*)

```
    010111        +23
  × 110110      ×  -10
  _____      _____
                  -230
```

```
                                   0  1  0  1  1  1
                                ×  0 -1 +1  0 -1  0
                              _____
                                                    0
              ┌  1  1  1  1  1│1  0  1  0  0  1
              │                              0
      sign    │  0  0  0│0  1  0  1  1  1
  extension   │  1₁1₁│1₁0₂1₁0₁0₁1
              └  _____
                 1  1  1  1  0  0  0  1  1  0  1  0
```

(*b*)

```
    110011         -13
  × 101100       ×  -20
  _____       _____
                   260
```

```
                                   1  1  0  0  1  1
                                ×  -1 +1  0 -1  0  0
                              _____
                                                    0
                                                 0
              ┌  0  0  0  0│0  0  1  1  0  1
              │                           0
      sign    │  1  1│1  1  0  0  1  1
  extension   │  0₁│0₁0₁1₁1₁1₁0₂1₁
              └  _____
                 0  0  0  1  0  0  0  0  0  1  0  0
```

(*c*)

```
    110101         -11
  × 011011       ×   27
  _____       _____
                   -297
```

```
                                   1  1  0  1  0  1
                                ×  +1  0 -1 +1  0 -1
              ┌  0  0  0  0  0  0│0  0  1  0  1  1
              │                              0
      sign    │  1  1  1  1│1  1  0  1  0  1
  extension   │  0  0  0│0  0  1  0  1  1
              │                        0
              │  1₁│1₁1₁0₁1₁0₁1₁   ₁
              └  _____
                 1  1  1  0  1  1  0  1  0  1  1  1
```

(*d*)

```
    001111          15
  × 001111       ×  15
  _____       _____
                   225
```

```
                                   0  0  1  1  1  1
                                ×  0 +1  0  0  0 -1
                     1  1  1  1  1  1  1  1  1  0  0  0  1
                     0  0  0  0  1  1  1  1
                  _____
                     0  0  0  0  1  1  1  0  0  0  0  1
```

17

6.18. The multiplication answers are:

(*a*)
```
   010111
 × 110110
```

```
                    0  1  0  1  1  1
                        -1    +2    -2
 1  1  1  1  1 │ 1  0  1  0  0  1  0
 0  0  0 │ 0  1  0  1  1  1  0
 1 │ 1₁ 1₁ 0₂ 1₁ 0₁ 0₁ 1
 1  1  1  1  0  0  0  1  1  0  1  0
```

(*b*)
```
   110011
 × 101100
```

```
                    1  1  0  0  1  1
                        -1    -1     0
                                     0
 0  0  0 │ 0  0  0  1  1  0  1
 0 │ 0  0  0₁ 1₁ 1₁ 0₁ 1
 0  0  0  1  0  0  0  0  0  1  0  0
```

(*c*)
```
   110101
 × 011011
```

```
                    1  1  0  1  0  1
                        +2    -1    -1
 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  0₁ 1     ₁
 1  1  1  0  1  1  0  1  0  1  1  1
```

(*d*)
```
   001111
 × 001111
```

```
                 0  0  1  1  1  1
                     +1          -1
 1  1  1  1  1  1  1  1  0  0  0  1
 0  0  0  0  1  1  1  1
 0  0  0  0  1  1  1  0  0  0  0  1
```

18

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.)

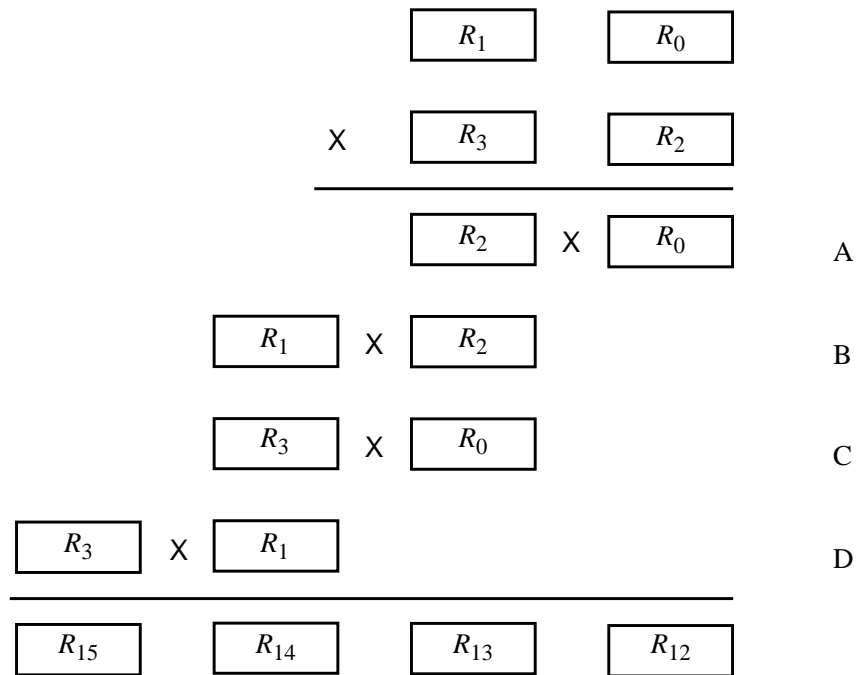Register A  (initially 0)

Shift right (Arithmetic)

| $a_n$ | $a_{n-1}$ | $\cdots$ | $a_0$ | | $q_{n-1}$ | $\cdots$ | $q_0$ | 0 |

Multiplier Q

ignore

sign extension bit

$n + 1$ bit adder

O ~ Nothing
M ~ Add M
$\overline{M}$ ~ Subtract M

MUX

00  Nothing
01  Add M
10  Subtract M
11  Nothing

Control sequencer

| $m_n$ | $m_{n-1}$ | $\cdots$ | $m_0$ |

Multiplicand M

0   Nothing
    Add M

1   Subtract M

19
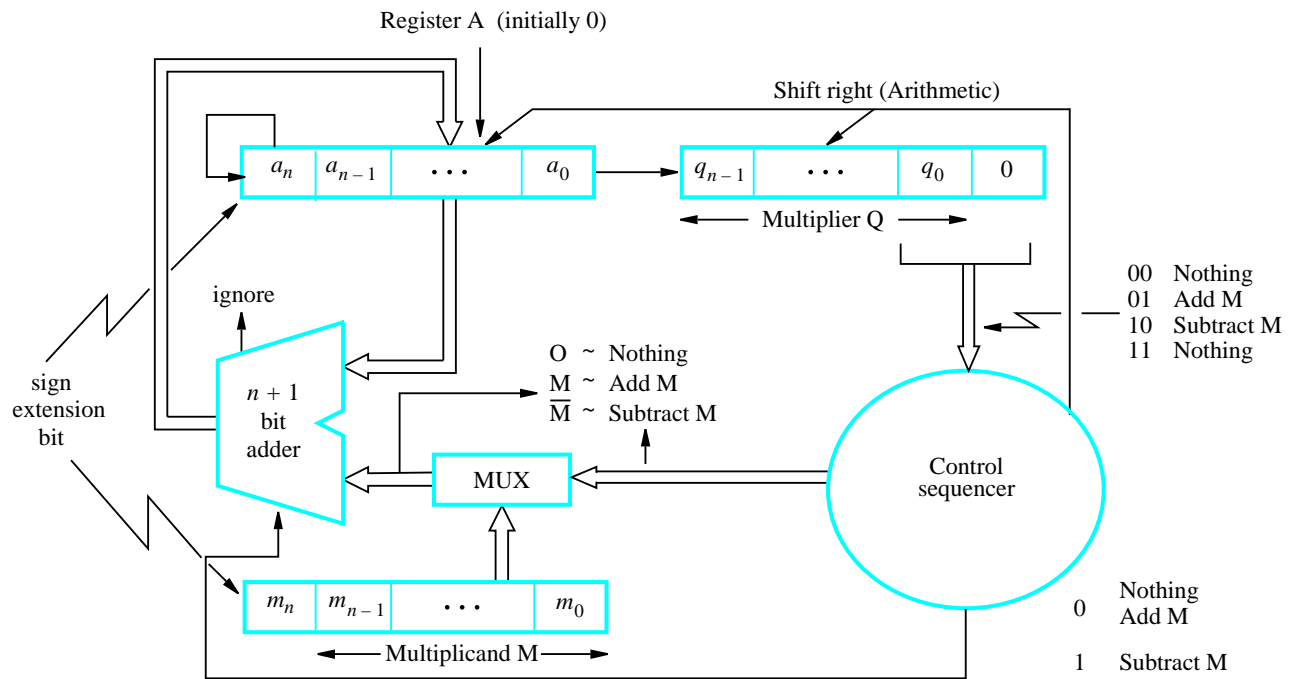
6.20 (a)

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

(b)

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

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:

$$
\begin{array}{cccccc}
 & & \boxed{R_1} & & \boxed{R_0} & \\
 & \times & \boxed{R_3} & & \boxed{R_2} & \\
\hline
 & & \boxed{R_2} & \times & \boxed{R_0} & \text{A} \\
 & \boxed{R_1} & \times & \boxed{R_2} & & \text{B} \\
 & \boxed{R_3} & \times & \boxed{R_0} & & \text{C} \\
\boxed{R_3} & \times & \boxed{R_1} & & & \text{D} \\
\hline
\boxed{R_{15}} & \boxed{R_{14}} & \boxed{R_{13}} & \boxed{R_{12}} & \\
\end{array}
$$

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

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

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

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

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

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

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

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

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

6.23. The number of reduction steps $n$ to reduce $k$ summands to 2 is given by $k(2/3)^n = 2$, because each step reduces 3 summands to 2. Then we have:
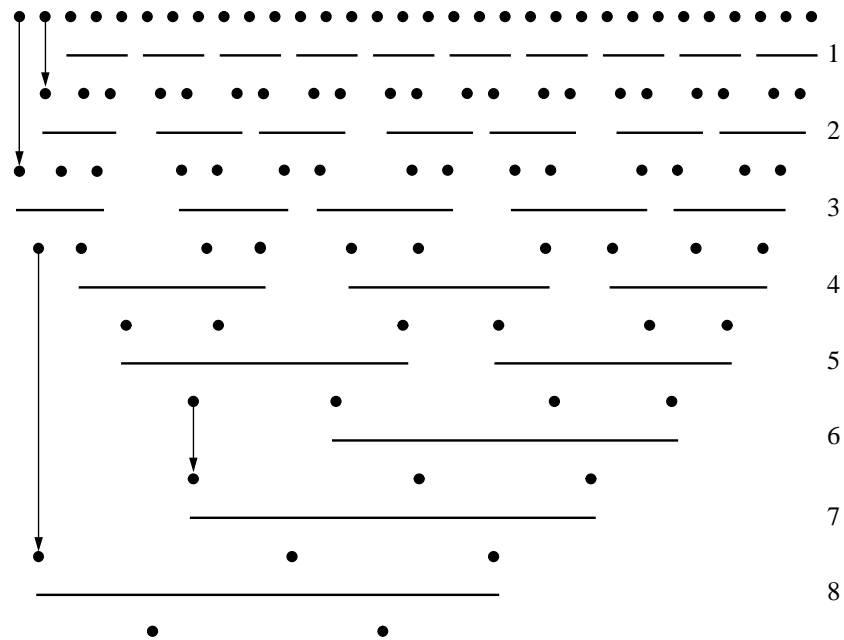
$$
\begin{aligned}
\log_2 k + n(\log_2 2 - \log_2 3) &= \log_2 2 \\
\log_2 k &= 1 + n(\log_2 3 - \log_2 2) \\
&= 1 + n(1.59 - 1) \\
n &= \frac{(\log_2 k) - 1}{0.59} \\
&= 1.7\log_2 k - 1.7
\end{aligned}
$$

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

6.24. (*a*) Six CSA levels are needed:
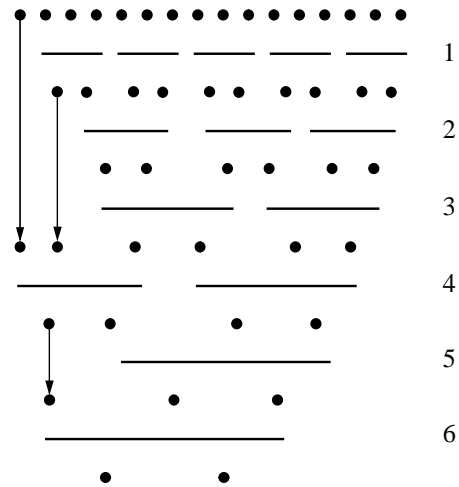


(*b*) Eight CSA levels are needed:



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

6.25. (a)

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

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

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

(d)

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

"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:

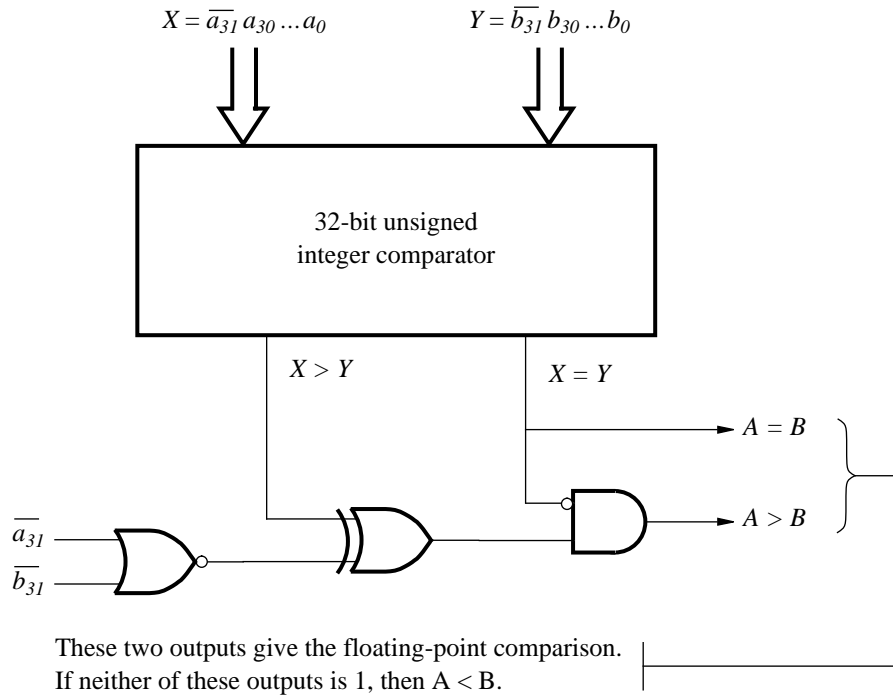| | |
|---|---|
| (A) | 1.11111111000 |
| (B) | 0.01001010101 |
| | 10.01001001101 |

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.



$$X = \overline{a_{31}}\,a_{30}\ldots a_0 \qquad Y = \overline{b_{31}}\,b_{30}\ldots b_0$$

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

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.

$$A \; = \; 0.100000$$
$$B \; = \; 0.100001$$

After shifting, we have:

| | |
|---|---|
| $A =$ | 0.100000 000 |
| $-B =$ | 0.000000 101   ⟵ sticky bit |
| | 0.011111 011 |
| normalize | 0.111110 110 |
| round | 0.111111       ⟵ correct answer (rounded) |

With only 2 guard bits, we would have had:

| | |
|---|---|
| $A =$ | 0.100000 00 |
| $-B =$ | 0.000000 11 |
| | 0.011111 01 |
| normalize | 0.111110 10 |
| round | 0.111110 |

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:

$$0.123 \; = \; 0.00011 \, 111$$
$$0.1 \; = \; 0.00011 \, 001$$

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



$S_A S_B$

| ADD(0)/ SUBTRACT(1) (AS) | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |

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

$S_B \, 8_s$  $AS \; S_A$

| | 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 |

$25_s = 1$

$$S_R = 25_s \, \overline{S_A} + \overline{25_s} \, S_A \, \overline{8_s} + AS \, \overline{S_B} \, 8_s + \overline{AS} \, S_B \, 8_s$$

28

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.

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

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:

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

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.

29

6.34. (*a*)
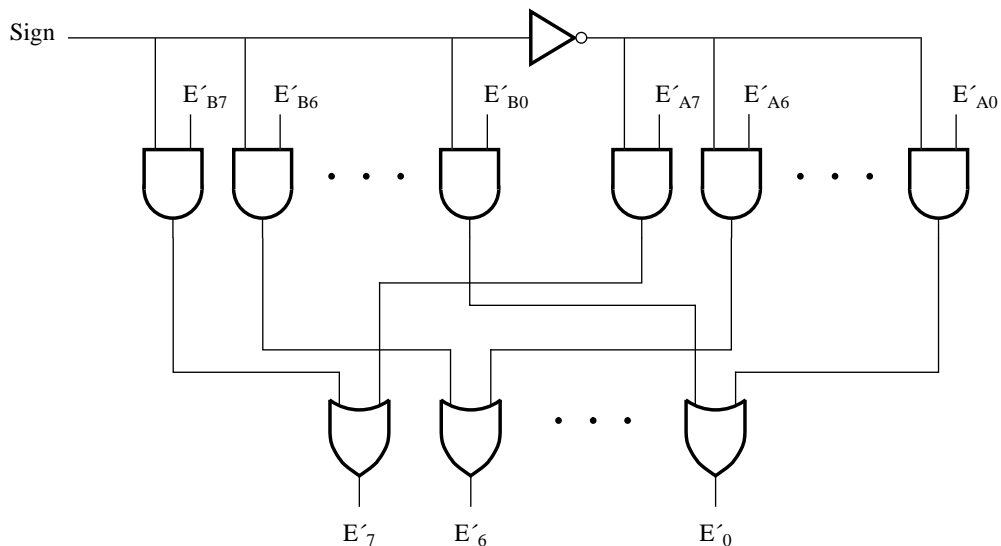


(*b*) The SWAP network is a pair of multiplexers, each one similar to (*a*).

6.35. Let $m = m_{24}m_{23} \ldots m_0$ 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:

$$
\begin{aligned}
z_{-1} &= m_{24} \\
z_0 &= \overline{m}_{24}m_{23} \\
z_1 &= \overline{m}_{24}\overline{m}_{23}m_{22} \\
&\phantom{=}\quad . \\
&\phantom{=}\quad . \\
&\phantom{=}\quad . \\
z_{23} &= \overline{m}_{24}\overline{m}_{23} \ldots m_0 \\
z_{24} &= \overline{m}_{24}\overline{m}_{23} \ldots \overline{m}_0
\end{aligned}
$$

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

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

The lookahead logic must generate the group input carries $c_0$, $c_4$, $c_8$, ..., $c_{24}$, 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:

$$
\begin{aligned}
c_0 &= G_6 + P_6 G_5 + \ldots + P_6 P_5 P_4 P_3 P_2 P_1 G_0 \\
c_4 &= G_0 + P_0 G_6 + P_0 P_6 G_5 + \ldots + P_0 P_6 P_5 P_4 P_3 P_2 G_1 \\
&\quad . \\
&\quad . \\
&\quad .
\end{aligned}
$$

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.