FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

COMPUTER ORGANIZATION
AND EMBEDDED SYSTEMS

SIXTH EDITION

Carl Hamacher
Queen's University

Zvonko Vranesic
University of Toronto

Safwat Zaky
University of Toronto

Naraig Manjikian
Queen's University

Mc Graw Hill
Connect Learn Succeed™

https://hemanthrajhemu.github.io

6.9 Superscalar Operation 212
6.9.1 Branches and Data Dependencies 214
6.9.2 Out-of-Order Execution 215
6.9.3 Execution Completion 216
6.9.4 Dispatch Operation 217
6.10 Pipelining in CISC Processors 218
6.10.1 Pipelining in ColdFire Processors 219
6.10.2 Pipelining in Intel Processors 219
6.11 Concluding Remarks 220
6.12 Examples of Solved Problems 220
Problems 222
References 226

Chapter 7
INPUT/OUTPUT ORGANIZATION 227

7.1 Bus Structure 228
7.2 Bus Operation 229
7.2.1 Synchronous Bus 230
7.2.2 Asynchronous Bus 233
7.2.3 Electrical Considerations 236
7.3 Arbitration 237
7.4 Interface Circuits 238
7.4.1 Parallel Interface 239
7.4.2 Serial Interface 243
7.5 Interconnection Standards 247
7.5.1 Universal Serial Bus (USB) 247
7.5.2 FireWire 251
7.5.3 PCI Bus 252
7.5.4 SCSI Bus 256
7.5.5 SATA 258
7.5.6 SAS 258
7.5.7 PCI Express 258
7.6 Concluding Remarks 260
7.7 Solved Problems 260
Problems 263
References 266

Chapter 8
THE MEMORY SYSTEM 267

8.1 Basic Concepts 268
8.2 Semiconductor RAM Memories 270
8.2.1 Internal Organization of Memory Chips 270
8.2.2 Static Memories 271
8.2.3 Dynamic RAMs 274
8.2.4 Synchronous DRAMs 276
8.2.5 Structure of Larger Memories 279
8.3 Read-only Memories 282
8.3.1 ROM 283
8.3.2 PROM 283
8.3.3 EPROM 284
8.3.4 EEPROM 284
8.3.5 Flash Memory 284
8.4 Direct Memory Access 285
8.5 Memory Hierarchy 288
8.6 Cache Memories 289
8.6.1 Mapping Functions 291
8.6.2 Replacement Algorithms 296
8.6.3 Examples of Mapping Techniques 297
8.7 Performance Considerations 300
8.7.1 Hit Rate and Miss Penalty 301
8.7.2 Caches on the Processor Chip 302
8.7.3 Other Enhancements 303
8.8 Virtual Memory 305
8.8.1 Address Translation 306
8.9 Memory Management Requirements 310
8.10 Secondary Storage 311
8.10.1 Magnetic Hard Disks 311
8.10.2 Optical Disks 317
8.10.3 Magnetic Tape Systems 322
8.11 Concluding Remarks 323
8.12 Solved Problems 324
Problems 328
References 332

Chapter 9
ARITHMETIC 335

9.1 Addition and Subtraction of Signed Numbers 336
9.1.1 Addition/Subtraction Logic Unit 336
9.2 Design of Fast Adders 339
9.2.1 Carry-Lookahead Addition 340
9.3 Multiplication of Unsigned Numbers 344
9.3.1 Array Multiplier 344
9.3.2 Sequential Circuit Multiplier 346
9.4 Multiplication of Signed Numbers 346
9.4.1 The Booth Algorithm 348
9.5 Fast Multiplication 351
9.5.1 Bit-Pair Recoding of Multipliers 352
9.5.2 Carry-Save Addition of Summands 353

https://hemanthrajhemu.github.io

9.5.3 Summand Addition Tree using 3-2 Reducers 355
9.5.4 Summand Addition Tree using 4-2 Reducers 357
9.5.5 Summary of Fast Multiplication 359
9.6 Integer Division 360
9.7 Floating-Point Numbers and Operations 363
9.7.1 Arithmetic Operations on Floating-Point Numbers 367
9.7.2 Guard Bits and Truncation 368
9.7.3 Implementing Floating-Point Operations 369
9.8 Decimal-to-Binary Conversion 372
9.9 Concluding Remarks 372
9.10 Solved Problems 374
Problems 377
References 383

Chapter 10
EMBEDDED SYSTEMS 385

10.1 Examples of Embedded Systems 386
10.1.1 Microwave Oven 386
10.1.2 Digital Camera 387
10.1.3 Home Telemetry 390
10.2 Microcontroller Chips for Embedded Applications 390
10.3 A Simple Microcontroller 392
10.3.1 Parallel I/O Interface 392
10.3.2 Serial I/O Interface 395
10.3.3 Counter/Timer 397
10.3.4 Interrupt-Control Mechanism 399
10.3.5 Programming Examples 399
10.4 Reaction Timer—A Complete Example 401
10.5 Sensors and Actuators 407
10.5.1 Sensors 407
10.5.2 Actuators 410
10.5.3 Application Examples 411
10.6 Microcontroller Families 412
10.6.1 Microcontrollers Based on the Intel 8051 413
10.6.2 Freescale Microcontrollers 413
10.6.3 ARM Microcontrollers 414
10.7 Design Issues 414
10.8 Concluding Remarks 417
Problems 418
References 420

Chapter 11
SYSTEM-ON-A-CHIP—A CASE STUDY 421

11.1 FPGA Implementation 422
11.1.1 FPGA Devices 423
11.1.2 Processor Choice 423
11.2 Computer-Aided Design Tools 424
11.2.1 Altera CAD Tools 425
11.3 Alarm Clock Example 428
11.3.1 User's View of the System 428
11.3.2 System Definition and Generation 429
11.3.3 Circuit Implementation 430
11.3.4 Application Software 431
11.4 Concluding Remarks 440
Problems 440
References 441

Chapter 12
PARALLEL PROCESSING AND PERFORMANCE 443

12.1 Hardware Multithreading 444
12.2 Vector (SIMD) Processing 445
12.2.1 Graphics Processing Units (GPUs) 448
12.3 Shared-Memory Multiprocessors 448
12.3.1 Interconnection Networks 450
12.4 Cache Coherence 453
12.4.1 Write-Through Protocol 453
12.4.2 Write-Back protocol 454
12.4.3 Snoopy Caches 454
12.4.4 Directory-Based Cache Coherence 456
12.5 Message-Passing Multicomputers 456
12.6 Parallel Programming for Multiprocessors 456
12.7 Performance Modeling 460
12.8 Concluding Remarks 461
Problems 462
References 463

Appendix A
LOGIC CIRCUITS 465

A.1 Basic Logic Functions
A.1.1 Electronic Logic Gates 469
A.2 Synthesis of Logic Functions 470

https://hemanthrajhemu.github.io

chapter

9

ARITHMETIC

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Adder and subtractor circuits
- High-speed adders based on carry-lookahead logic circuits
- The Booth algorithm for multiplication of signed numbers
- High-speed multipliers based on carry-save addition
- Logic circuits for division
- Arithmetic operations on floating-point numbers conforming to the IEEE standard

https://hemanthrajhemu.github.io

Addition and subtraction of two numbers are basic operations at the machine-instruction level in all computers. These operations, as well as other arithmetic and logic operations, are implemented in the arithmetic and logic unit (ALU) of the processor. In this chapter, we present the logic circuits used to implement arithmetic operations. The time needed to perform addition or subtraction affects the processor's performance. Multiply and divide operations, which require more complex circuitry than either addition or subtraction operations, also affect performance. We present some of the techniques used in modern computers to perform arithmetic operations at high speed. Operations on floating-point numbers are also described.

In Section 1.4 of Chapter 1, we described the representation of signed binary numbers, and showed that 2's-complement is the best representation from the standpoint of performing addition and subtraction operations. The examples in Figure 1.6 show that two, n-bit, signed numbers can be added using n-bit binary addition, treating the sign bit the same as the other bits. In other words, a logic circuit that is designed to add unsigned binary numbers can also be used to add signed numbers in 2's-complement. The first two sections of this chapter present logic circuits for addition and subtraction.

9.1 Addition and Subtraction of Signed Numbers

Figure 9.1 shows the truth table for the sum and carry-out functions for adding equally weighted bits x_i and y_i in two numbers X and Y. The figure also shows logic expressions for these functions, along with an example of addition of the 4-bit unsigned numbers 7 and 6. Note that each stage of the addition process must accommodate a carry-in bit. We use c_i to represent the carry-in to stage i, which is the same as the carry-out from stage $(i - 1)$.

The logic expression for s_i in Figure 9.1 can be implemented with a 3-input XOR gate, used in Figure 9.2*a* as part of the logic required for a single stage of binary addition. The carry-out function, c_{i+1}, is implemented with an AND-OR circuit, as shown. A convenient symbol for the complete circuit for a single stage of addition, called a *full adder* (FA), is also shown in the figure.

A cascaded connection of n full-adder blocks can be used to add two n-bit numbers, as shown in Figure 9.2*b*. Since the carries must propagate, or ripple, through this cascade, the configuration is called a *ripple-carry adder*.

The carry-in, c_0, into the *least-significant-bit* (LSB) position provides a convenient means of adding 1 to a number. For instance, forming the 2's-complement of a number involves adding 1 to the 1's-complement of the number. The carry signals are also useful for interconnecting k adders to form an adder capable of handling input numbers that are kn bits long, as shown in Figure 9.2*c*.

9.1.1 Addition/Subtraction Logic Unit

The n-bit adder in Figure 9.2*b* can be used to add 2's-complement numbers X and Y, where the x_{n-1} and y_{n-1} bits are the sign bits. The carry-out bit c_n is not part of the answer. Arithmetic overflow was discussed in Section 1.4. It occurs when the signs of the two

https://hemanthrajhemu.github.io

x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \overline{x}_i\overline{y}_i c_i + \overline{x}_i y_i \overline{c}_i + x_i \overline{y}_i \overline{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$
$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

Example:

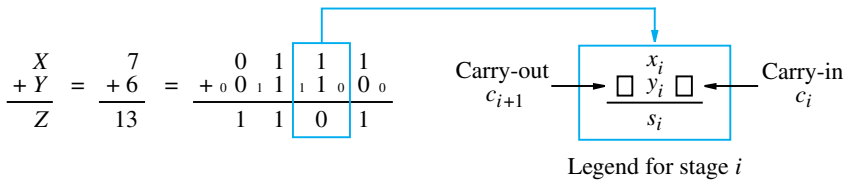


Figure 9.1 Logic specification for a stage of binary addition.

operands are the same, but the sign of the result is different. Therefore, a circuit to detect overflow can be added to the n-bit adder by implementing the logic expression

$$\text{Overflow} = x_{n-1}y_{n-1}\overline{s}_{n-1} + \overline{x}_{n-1}\overline{y}_{n-1}s_{n-1}$$

It can also be shown that overflow occurs when the carry bits c_n and c_{n-1} are different. (See Problem 9.5.) Therefore, a simpler circuit for detecting overflow can be obtained by implementing the expression $c_n \oplus c_{n-1}$ with an XOR gate.

In order to perform the subtraction operation $X - Y$ on 2's-complement numbers X and Y, we form the 2's-complement of Y and add it to X. The logic circuit shown in Figure 9.3 can be used to perform either addition or subtraction based on the value applied to the Add/Sub input control line. This line is set to 0 for addition, applying Y unchanged to one of the adder inputs along with a carry-in signal, c_0, of 0. When the Add/Sub control line is set to 1, the Y number is 1's-complemented (that is, bit-complemented) by the XOR gates and c_0 is set to 1 to complete the 2's-complementation of Y. Recall that 2's-complementing a negative number is done in exactly the same manner as for a positive number. An XOR gate can be added to Figure 9.3 to detect the overflow condition $c_n \oplus c_{n-1}$.

https://hemanthrajhemu.github.io

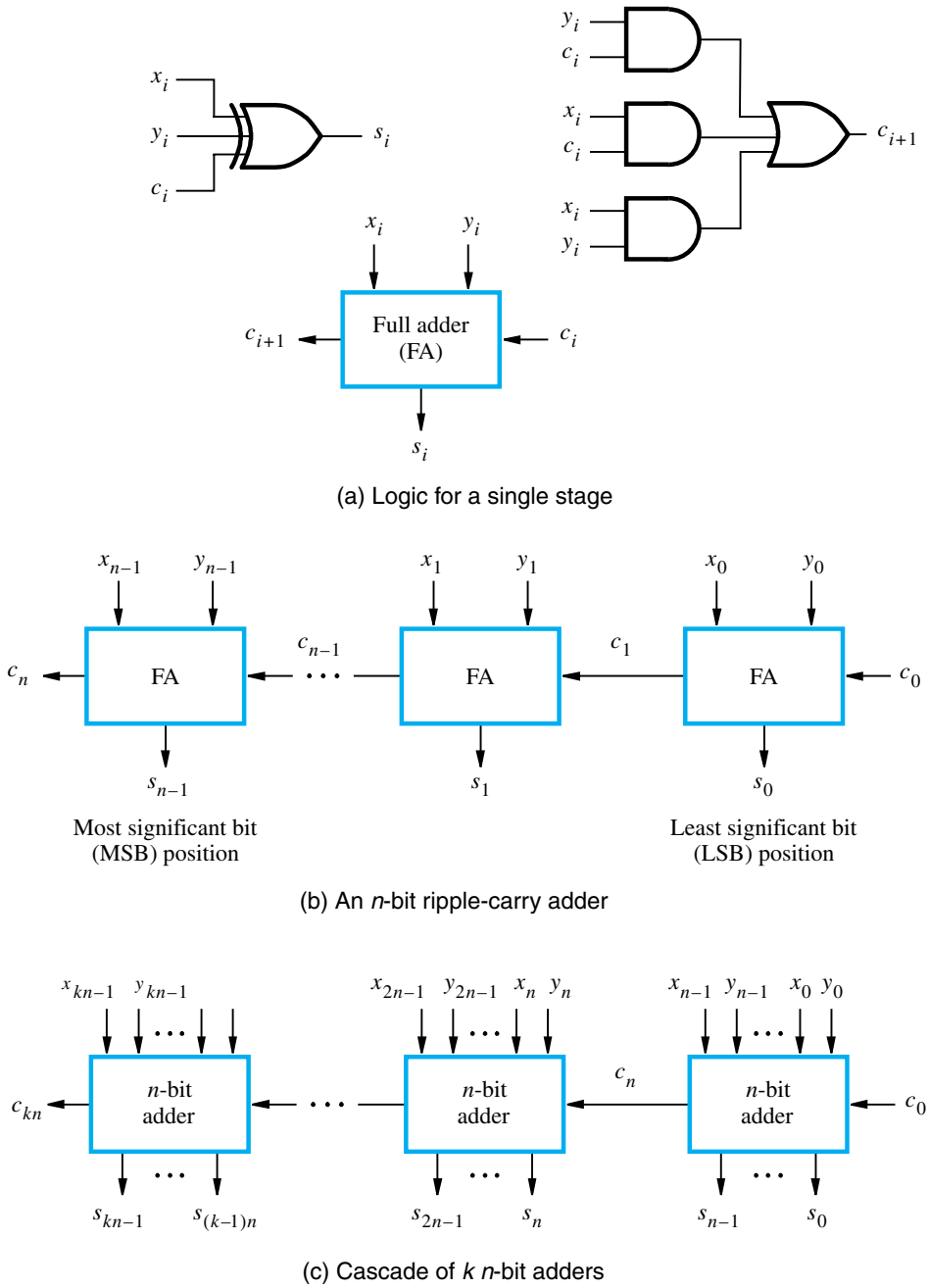


Figure 9.2 Logic for addition of binary numbers.

https://hemanthrajhemu.github.io

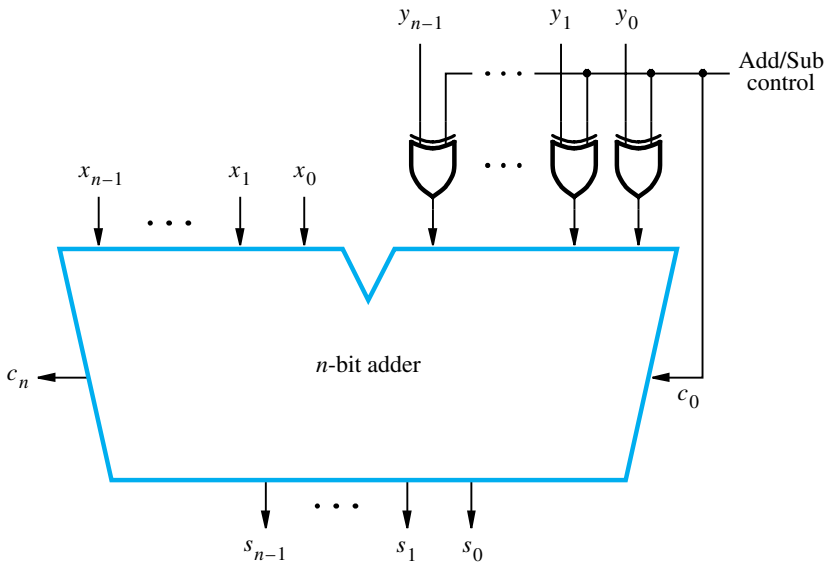


Figure 9.3 Binary addition/subtraction logic circuit.

9.2 DESIGN OF FAST ADDERS

If an n-bit ripple-carry adder is used in the addition/subtraction circuit of Figure 9.3, it may have too much delay in developing its outputs, s_0 through s_{n-1} and c_n. Whether or not the delay incurred is acceptable can be decided only in the context of the speed of other processor components and the data transfer times of registers and cache memories. The delay through a network of logic gates depends on the integrated circuit electronic technology used in fabricating the network and on the number of gates in the paths from inputs to outputs. The delay through any combinational circuit constructed from gates in a particular technology is determined by adding up the number of logic-gate delays along the longest signal propagation path through the circuit. In the case of the n-bit ripple-carry adder, the longest path is from inputs x_0, y_0, and c_0 at the LSB position to outputs c_n and s_{n-1} at the *most-significant-bit* (MSB) position.

Using the implementation indicated in Figure 9.2a, c_{n-1} is available in $2(n-1)$ gate delays, and s_{n-1} is correct one XOR gate delay later. The final carry-out, c_n, is available after $2n$ gate delays. Therefore, if a ripple-carry adder is used to implement the addition/subtraction unit shown in Figure 9.3, all sum bits are available in $2n$ gate delays, including the delay through the XOR gates on the Y input. Using the implementation $c_n \oplus c_{n-1}$ for overflow, this indicator is available after $2n + 2$ gate delays.

Two approaches can be taken to reduce delay in adders. The first approach is to use the fastest possible electronic technology. The second approach is to use a logic gate network called a carry-lookahead network, which is described in the next section.

https://hemanthrajhemu.github.io

9.2.1 CARRY-LOOKAHEAD ADDITION

A fast adder circuit must speed up the generation of the carry signals. The logic expressions for s_i (sum) and c_{i+1} (carry-out) of stage i (see Figure 9.1) are

$$s_i = x_i \oplus y_i \oplus c_i$$

and

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Factoring the second equation into

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

we can write

$$c_{i+1} = G_i + P_i c_i$$

where

$$G_i = x_i y_i \qquad \text{and} \qquad P_i = x_i + y_i$$

The expressions G_i and P_i are called the *generate* and *propagate* functions for stage i. If the generate function for stage i is equal to 1, then $c_{i+1} = 1$, independent of the input carry, c_i. This occurs when both x_i and y_i are 1. The propagate function means that an input carry will produce an output carry when either x_i is 1 or y_i is 1. All G_i and P_i functions can be formed independently and in parallel in one logic-gate delay after the X and Y operands are applied to the inputs of an n-bit adder. Each bit stage contains an AND gate to form G_i, an OR gate to form P_i, and a three-input XOR gate to form s_i. A simpler circuit can be derived by observing that an adequate propagate function can be realized as $P_i = x_i \oplus y_i$, which differs from $P_i = x_i + y_i$ only when $x_i = y_i = 1$. But, in this case $G_i = 1$, so it does not matter whether P_i is 0 or 1. Then, using a cascade of two 2-input XOR gates to realize the 3-input XOR function for s_i, the basic B cell in Figure 9.4*a* can be used in each bit stage.

Expanding c_i in terms of $i - 1$ subscripted variables and substituting into the c_{i+1} expression, we obtain

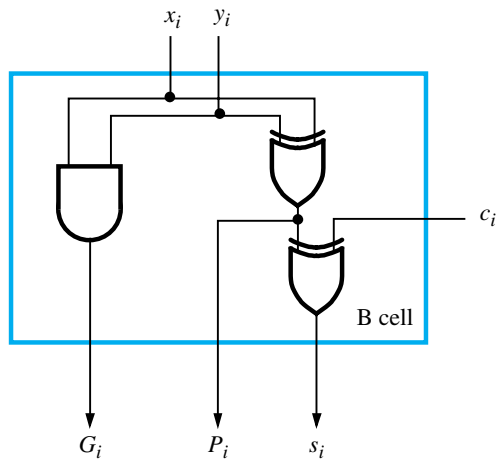
$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1}$$

Continuing this type of expansion, the final expression for any carry variable is

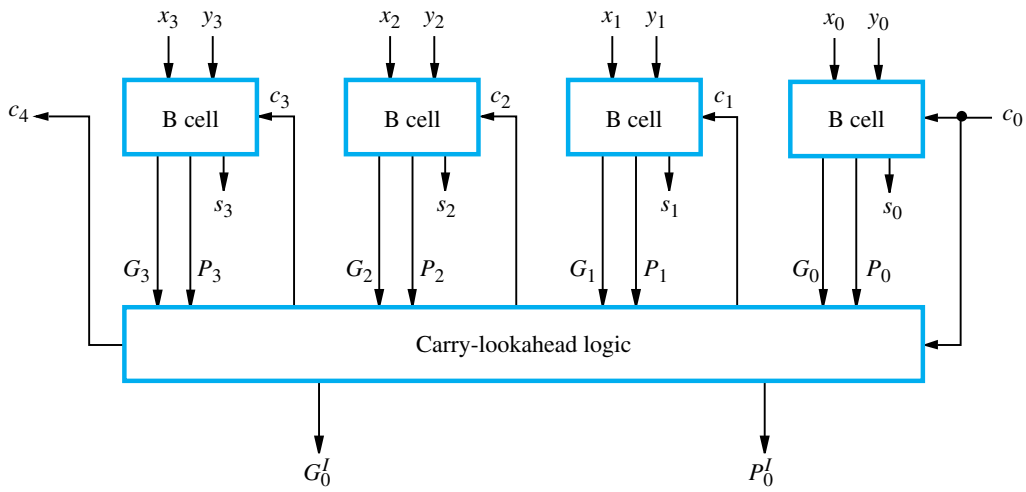
$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots + P_i P_{i-1} \cdots P_1 G_0 + P_i P_{i-1} \cdots P_0 c_0 \qquad (9.1)$$

Thus, all carries can be obtained three gate delays after the input operands X, Y, and c_0 are applied because only one gate delay is needed to develop all P_i and G_i signals, followed by two gate delays in the AND-OR circuit for c_{i+1}. After a further XOR gate delay, all sum bits are available. In total, the n-bit addition process requires only four gate delays, independent of n.

https://hemanthrajhemu.github.io



(a) Bit-stage cell



(b) 4-bit adder

Figure 9.4 A 4-bit carry-lookahead adder.

Let us consider the design of a 4-bit adder. The carries can be implemented as

$$c_1 = G_0 + P_0 c_0$$
$$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$
$$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$
$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

https://hemanthrajhemu.github.io

The complete 4-bit adder is shown in Figure 9.4*b*. The carries are produced in the block labeled carry-lookahead logic. An adder implemented in this form is called a *carry-lookahead adder*. Delay through the adder is 3 gate delays for all carry bits and 4 gate delays for all sum bits. In comparison, a 4-bit ripple-carry adder requires 7 gate delays for s_3 and 8 gate delays for c_4.

If we try to extend the carry-lookahead adder design of Figure 9.4*b* for longer operands, we encounter the problem of gate fan-in constraints. From Expression 9.1, we see that the last AND gate and the OR gate require a fan-in of $i + 2$ in generating c_{i+1}. A fan-in of 5 is required for c_4 in the 4-bit adder. This is about the limit for practical gates. So the adder design shown in Figure 9.4*b* cannot be extended easily for longer operands. However, it is possible to build longer adders by cascading a number of 4-bit adders, as shown in Figure 9.2*c*.

Eight, 4-bit, carry-lookahead adders can be connected as in Figure 9.2*c* to form a 32-bit adder. The delays in generating sum bits s_{31}, s_{30}, s_{29}, s_{28}, and carry bit c_{32} in the high-order 4-bit adder in this cascade are calculated as follows. The carry-out c_4 from the low-order adder is available 3 gate delays after the input operands X, Y, and c_0 are applied to the 32-bit adder. Then, c_8 is available at the output of the second adder after a further 2 gate delays, c_{12} is available after a further 2 gate delays, and so on. Finally, c_{28}, the carry-in to the high-order 4-bit adder, is available after a total of $(6 \times 2) + 3 = 15$ gate delays. Then, c_{32} and all carries inside the high-order adder are available after a further 2 gate delays, and all 4 sum bits are available after 1 more gate delay, for a total of 18 gate delays. This should be compared to total delays of 63 and 64 for s_{31} and c_{32} if a ripple-carry adder is used.

In the next section, we show how it is possible to improve upon the cascade structure just discussed, leading to further reduction in adder delay. The key idea is to generate the carries c_4, c_8, ... in parallel, similar to the way that c_1, c_2, c_3, and c_4, are generated in parallel in the 4-bit carry-lookahead adder.

Higher-Level Generate and Propagate Functions

In the 32-bit adder just discussed, the carries $c_4, c_8, c_{12}, \ldots$ ripple through the 4-bit adder blocks with two gate delays per block, analogous to the way that individual carries ripple through each bit stage in a ripple-carry adder. It is possible to use the lookahead approach to develop the carries c_4, c_8, c_{12}, ... in parallel by using higher-level block generate and propagate functions.

Figure 9.5 shows a 16-bit adder built from four 4-bit adder blocks. These blocks provide new output functions defined as G_k^I and P_k^I, where $k = 0$ for the first 4-bit block, $k = 1$ for the second 4-bit block, and so on, as shown in Figures 9.4*b* and 9.5. In the first block,

$$P_0^I = P_3P_2P_1P_0$$

and

$$G_0^I = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$

The first-level G_i and P_i functions determine whether bit stage i generates or propagates a carry. The second-level G_k^I and P_k^I functions determine whether block k generates or propagates a carry. With these new functions available, it is not necessary to wait for carries to ripple through the 4-bit blocks. Carry c_{16} is formed by one of the carry-lookahead

https://hemanthrajhemu.github.io

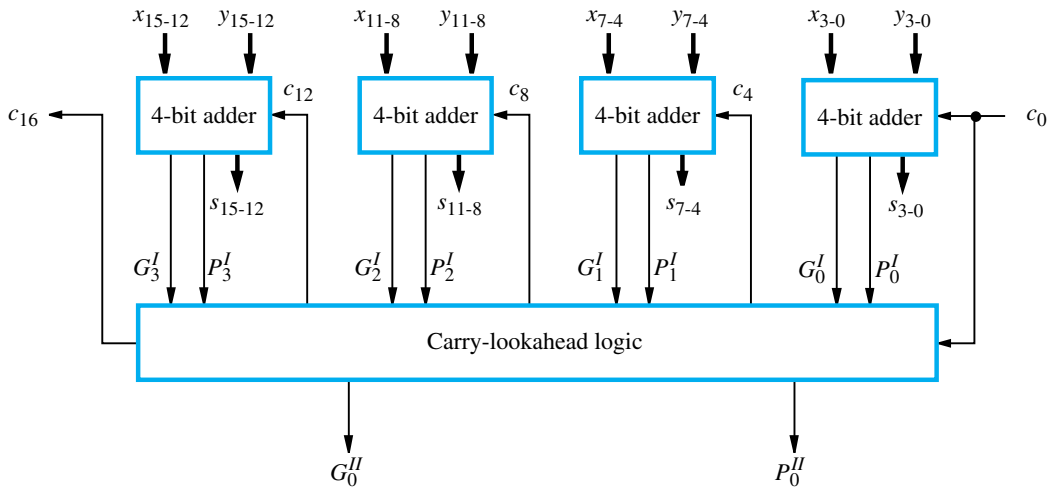


Figure 9.5 A 16-bit carry-lookahead adder built from 4-bit adders (see Figure 9.4*b*).

circuits in Figure 9.5 as

$$c_{16} = G_3^I + P_3^I G_2^I + P_3^I P_2^I G_1^I + P_3^I P_2^I P_1^I G_0^I + P_3^I P_2^I P_1^I P_0^I c_0$$

The input carries to the 4-bit blocks are formed in parallel by similar shorter expressions. Expressions for c_{16}, c_{12}, c_8, and c_4, are identical in form to the expressions for c_4, c_3, c_2, and c_1, respectively, implemented in the carry-lookahead circuits in Figure 9.4*b*. Only the variable names are different. Therefore, the structure of the carry-lookahead circuits in Figure 9.5 is identical to the carry-lookahead circuits in Figure 9.4*b*. However, the carries c_4, c_8, c_{12}, and c_{16}, generated internally by the 4-bit adder blocks, are not needed in Figure 9.5 because they are generated by the higher-level carry-lookahead circuits.

Now, consider the delay in producing outputs from the 16-bit carry-lookahead adder. The delay in developing the carries produced by the carry-lookahead circuits is two gate delays more than the delay needed to develop the G_k^I and P_k^I functions. The latter require two gate delays and one gate delay, respectively, after the generation of G_i and P_i. Therefore, all carries produced by the carry-lookahead circuits are available 5 gate delays after X, Y, and c_0 are applied as inputs. The carry c_{15} is generated inside the high-order 4-bit block in Figure 9.5 in two gate delays after c_{12}, followed by s_{15} in one further gate delay. Therefore, s_{15} is available after 8 gate delays. If a 16-bit adder is built by cascading 4-bit carry-lookahead adder blocks, the delays in developing c_{16} and s_{15} are 9 and 10 gate delays, respectively, as compared to 5 and 8 gate delays for the configuration in Figure 9.5.

Two 16-bit adder blocks can be cascaded to implement a 32-bit adder. In this configuration, the output c_{16} from the low-order block is the carry input to the high-order block. The delay is much lower than the delay through the 32-bit adder that we discussed earlier, which was built by cascading eight 4-bit adders. In that configuration, recall that s_{31} is available after 18 gate delays and c_{32} is available after 17 gate delays. The delay analysis

https://hemanthrajhemu.github.io

for the cascade of two 16-bit adders is as follows. The carry c_{16} out of the low-order block is available after 5 gate delays, as calculated above. Then, both c_{28} and c_{32} are available in the high-order block after a further 2 gate delays, and c_{31} is available 2 gate delays after c_{28}. Therefore, c_{31} is available after a total of 9 gate delays, and s_{31} is available in 10 gate delays. Recapitulating, s_{31} and c_{32} are available after 10 and 7 gate delays, respectively, compared to 18 and 17 gate delays for the same outputs if the 32-bit adder is built from a cascade of eight 4-bit adders.

The same reasoning used in developing second-level G_k^I and P_k^I functions from first-level G_i and P_i functions can be used to develop third-level G_k^{II} and P_k^{II} functions from G_k^I and P_k^I functions. Two such third-level functions are shown as outputs from the carry-lookahead logic in Figure 9.5. A 64-bit adder can be built from four of the 16-bit adders shown in Figure 9.5, along with additional carry-lookahead logic circuits that produce carries c_{16}, c_{32}, c_{48}, and c_{64}. Delay through this adder can be shown to be 12 gate delays for s_{63} and 7 gate delays for c_{64}, using an extension of the reasoning used above for the 16-bit adder. (See Problem 9.7.)

9.3 MULTIPLICATION OF UNSIGNED NUMBERS

The usual algorithm for multiplying integers by hand is illustrated in Figure 9.6*a* for the binary system. The product of two, unsigned, n-digit numbers can be accommodated in $2n$ digits, so the product of the two 4-bit numbers in this example is accommodated in 8 bits, as shown. In the binary system, multiplication of the multiplicand by one bit of the multiplier is easy. If the multiplier bit is 1, the multiplicand is entered in the appropriate shifted position. If the multiplier bit is 0, then 0s are entered, as in the third row of the example. The product is computed one bit at a time by adding the bit columns from right to left and propagating carry values between columns.

9.3.1 ARRAY MULTIPLIER

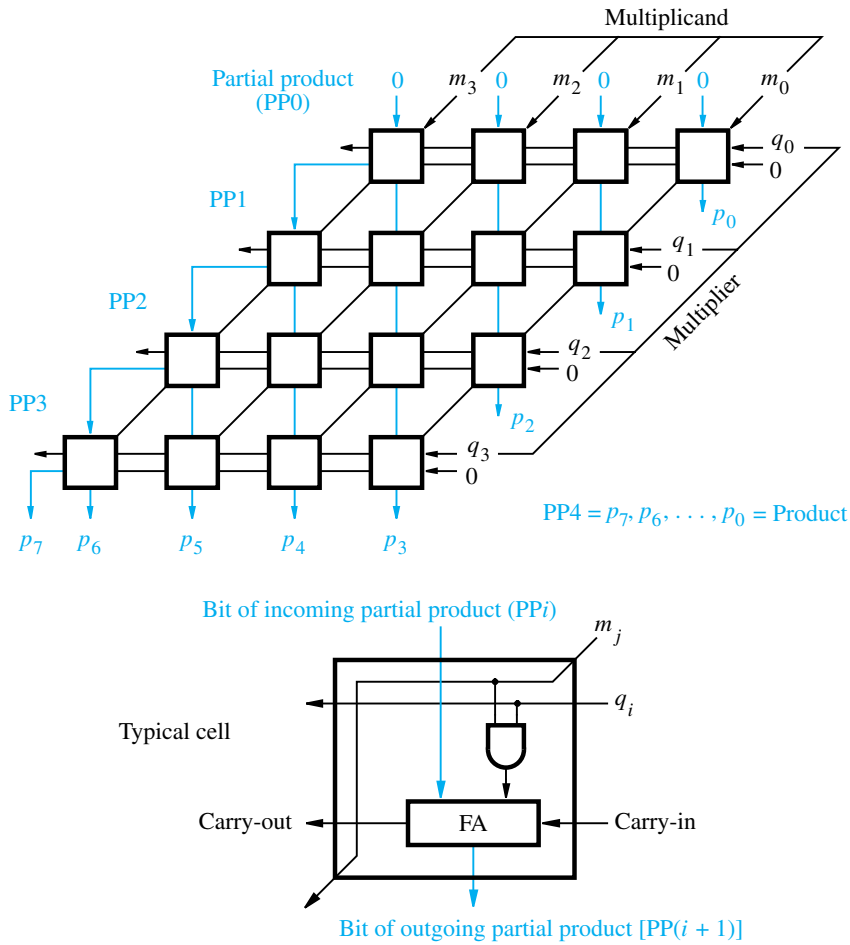
Binary multiplication of unsigned operands can be implemented in a combinational, two-dimensional, logic array, as shown in Figure 9.6*b* for the 4-bit operand case. The main component in each cell is a full adder, FA. The AND gate in each cell determines whether a multiplicand bit, m_j, is added to the incoming partial-product bit, based on the value of the multiplier bit, q_i. Each row i, where $0 \leq i \leq 3$, adds the multiplicand (appropriately shifted) to the incoming partial product, PPi, to generate the outgoing partial product, PP$(i+1)$, if $q_i = 1$. If $q_i = 0$, PPi is passed vertically downward unchanged. PP0 is all 0s, and PP4 is the desired product. The multiplicand is shifted left one position per row by the diagonal signal path. We note that the row-by-row addition done in the array circuit differs from the usual hand addition described previously, which is done column-by-column.

The worst-case signal propagation delay path is from the upper right corner of the array to the high-order product bit output at the bottom left corner of the array. This critical path consists of the staircase pattern that includes the two cells at the right end of each

https://hemanthrajhemu.github.io

```
                1 1 0 1      (13) Multiplicand M
              × 1 0 1 1      (11) Multiplier Q
              ---------
                1 1 0 1
              1 1 0 1
            0 0 0 0
          1 1 0 1
        ---------------
        1 0 0 0 1 1 1 1      (143) Product P
```

(a) Manual multiplication algorithm



(b) Array implementation

Figure 9.6 Array multiplication of unsigned binary operands.

https://hemanthrajhemu.github.io

row, followed by all the cells in the bottom row. Assuming that there are two gate delays from the inputs to the outputs of a full-adder block, FA, the critical path has a total of $6(n-1)-1$ gate delays, including the initial AND gate delay in all cells, for an $n \times n$ array. (See Problem 9.8.) In the first row of the array, no full adders are needed, because the incoming partial product PP0 is zero. This has been taken into account in developing the delay expression.

9.3.2 Sequential Circuit Multiplier

The combinational array multiplier just described uses a large number of logic gates for multiplying numbers of practical size, such as 32- or 64-bit numbers. Multiplication of two n-bit numbers can also be performed in a sequential circuit that uses a single n-bit adder.

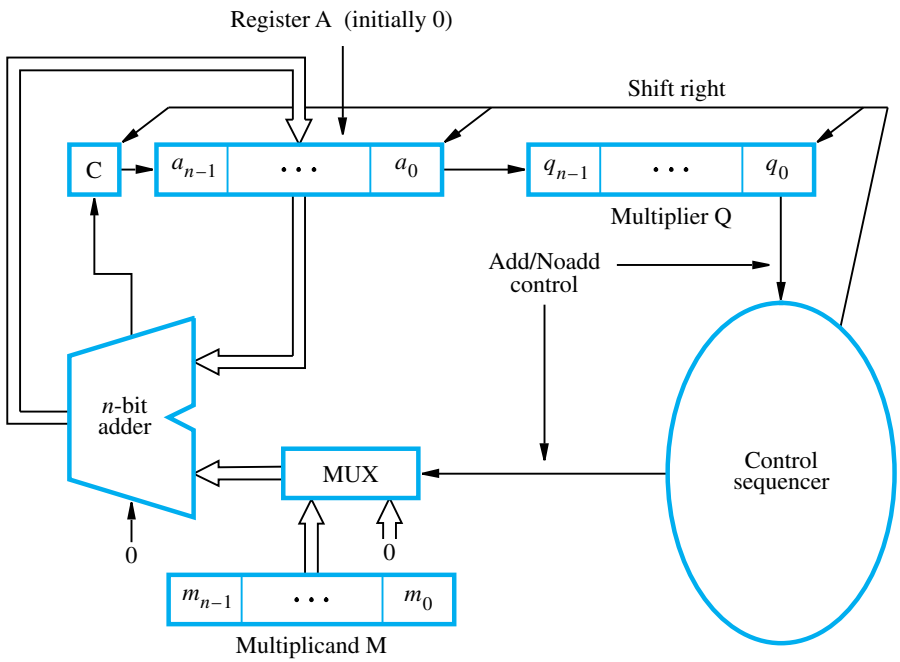
The block diagram in Figure 9.7*a* shows the hardware arrangement for sequential multiplication. This circuit performs multiplication by using a single n-bit adder n times to implement the spatial addition performed by the n rows of ripple-carry adders in Figure 9.6*b*. Registers A and Q are shift registers, concatenated as shown. Together, they hold partial product PPi while multiplier bit q_i generates the signal Add/Noadd. This signal causes the multiplexer MUX to select 0 when $q_i = 0$, or to select the multiplicand M when $q_i = 1$, to be added to PPi to generate PP$(i+1)$. The product is computed in n cycles. The partial product grows in length by one bit per cycle from the initial vector, PP0, of n 0s in register A. The carry-out from the adder is stored in flip-flop C, shown at the left end of register A. At the start, the multiplier is loaded into register Q, the multiplicand into register M, and C and A are cleared to 0. At the end of each cycle, C, A, and Q are shifted right one bit position to allow for growth of the partial product as the multiplier is shifted out of register Q. Because of this shifting, multiplier bit q_i appears at the LSB position of Q to generate the Add/Noadd signal at the correct time, starting with q_0 during the first cycle, q_1 during the second cycle, and so on. After they are used, the multiplier bits are discarded by the right-shift operation. Note that the carry-out from the adder is the leftmost bit of PP$(i+1)$, and it must be held in the C flip-flop to be shifted right with the contents of A and Q. After n cycles, the high-order half of the product is held in register A and the low-order half is in register Q. The multiplication example of Figure 9.6*a* is shown in Figure 9.7*b* as it would be performed by this hardware arrangement.

9.4 Multiplication of Signed Numbers

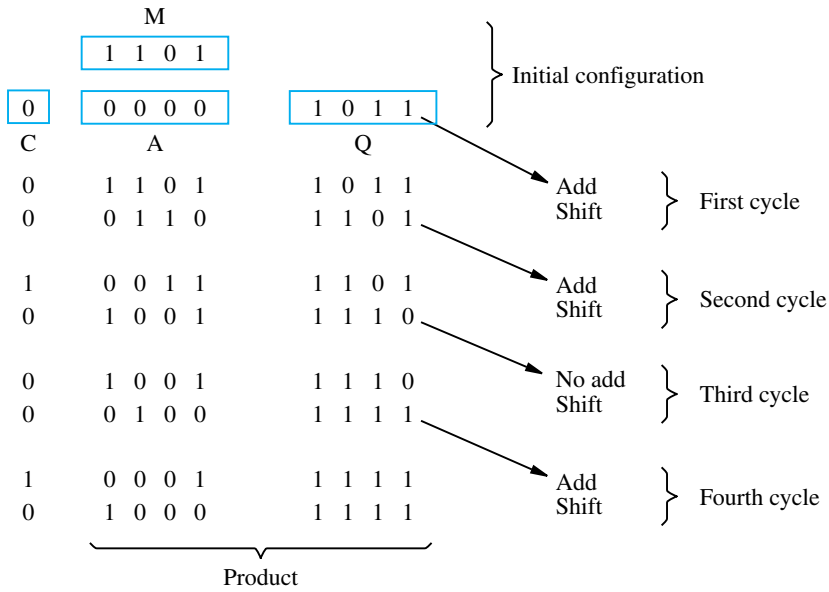
We now discuss multiplication of 2's-complement operands, generating a double-length product. The general strategy is still to accumulate partial products by adding versions of the multiplicand as selected by the multiplier bits.

First, consider the case of a positive multiplier and a negative multiplicand. When we add a negative multiplicand to a partial product, we must extend the sign-bit value of the multiplicand to the left as far as the product will extend. Figure 9.8 shows an example in which a 5-bit signed operand, -13, is the multiplicand. It is multiplied by $+11$ to get

https://hemanthrajhemu.github.io



(a) Register configuration



(b) Multiplication example

Figure 9.7 Sequential circuit binary multiplier.

https://hemanthrajhemu.github.io

						1	0	0	1	1	(−13)
					×	0	1	0	1	1	(+11)
	1	1	1	1	1	1	0	0	1	1	
	1	1	1	1	1	0	0	1	1		
Sign extension is shown in blue	0	0	0	0	0	0	0	0			
	1	1	1	0	0	1	1				
	0	0	0	0	0	0					
	1	1	0	1	1	1	0	0	0	1	(−143)

Figure 9.8 Sign extension of negative multiplicand.

the 10-bit product, −143. The sign extension of the multiplicand is shown in blue. The hardware discussed earlier can be used for negative multiplicands if it is augmented to provide for sign extension of the partial products.

For a negative multiplier, a straightforward solution is to form the 2's-complement of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier. This is possible because complementation of both operands does not change the value or the sign of the product. A technique that works equally well for both negative and positive multipliers, called the Booth algorithm, is described next.

9.4.1 THE BOOTH ALGORITHM

The Booth algorithm [1] generates a $2n$-bit product and treats both positive and negative 2's-complement n-bit operands uniformly. To understand the basis of this algorithm, consider a multiplication operation in which the multiplier is positive and has a single block of 1s, for example, 0011110. To derive the product, we could add four appropriately shifted versions of the multiplicand, as in the standard procedure. However, we can reduce the number of required operations by regarding this multiplier as the difference between two numbers:

$$\begin{array}{rcl} & 0100000 & (32) \\ - & 0000010 & (2) \\ \hline & 0011110 & (30) \end{array}$$

This suggests that the product can be generated by adding 2^5 times the multiplicand to the 2's-complement of 2^1 times the multiplicand. For convenience, we can describe the sequence of required operations by recoding the preceding multiplier as 0 +1 0 0 0 −1 0.

In general, in the Booth algorithm, −1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from

https://hemanthrajhemu.github.io

1 to 0, as the multiplier is scanned from right to left. Figure 9.9 illustrates the normal and the Booth algorithms for the example just discussed. The Booth algorithm clearly extends to any number of blocks of 1s in a multiplier, including the situation in which a single 1 is considered a block. Figure 9.10 shows another example of recoding a multiplier. The case when the least significant bit of the multiplier is 1 is handled by assuming that an implied 0 lies to its right. The Booth algorithm can also be used directly for negative multipliers, as shown in Figure 9.11.

To demonstrate the correctness of the Booth algorithm for negative multipliers, we use the following property of negative-number representations in the 2's-complement system.

							0	1	0	1	1	0	1
							0	0 + 1	+ 1	+ 1	+ 1	0	
							0	0	0	0	0	0	0
						0	1	0	1	1	0	1	
					0	1	0	1	1	0	1		
				0	1	0	1	1	0	1			
			0	0	0	0	0	0	0				
		0	0	0	0	0	0						
0	0	0	1	0	1	0	1	0	0	0	1	1	0

							0	1	0	1	1	0	1	
							0	+ 1	0	0	0	– 1	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	1	1	1	1	1	1	0	1	0	0	1	1		2's complement of the multiplicand
0	0	0	0	0	0	0	0	0	0	0	0			
0	0	0	0	0	0	0	0	0	0	0				
0	0	0	0	0	0	0	0	0	0					
0	0	0	1	0	1	1	0	1						
0	0	0	0	0	0	0	0							
0	0	0	1	0	1	0	1	0	0	0	1	1	0	

Figure 9.9 Normal and Booth multiplication schemes.

0	0	1	0	1	1	0	0	1	1	1	0	1	0	1	1	0	0
								⇓									
0	+ 1	– 1	+ 1	0	– 1	0	+ 1	0	0	– 1	+ 1	– 1	+ 1	0	– 1	0	0

Figure 9.10 Booth recoding of a multiplier.

https://hemanthrajhemu.github.io

$$
\begin{array}{rrrrrrl}
 & 0 & 1 & 1 & 0 & 1 & (+13) \\
\times & 1 & 1 & 0 & 1 & 0 & (-6) \\
\hline
\end{array}
\quad \Longrightarrow \quad
\begin{array}{rrrrrrrrrrl}
 & & & & & 0 & 1 & 1 & 0 & 1 & \\
 & & & & & 0 & -1 & +1 & -1 & 0 & \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & & \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & & & \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
\hline
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & (-78)
\end{array}
$$

Figure 9.11 Booth multiplication with a negative multiplier.

Suppose that the leftmost 0 of a negative number, X, is at bit position k, that is,

$$X = 11\ldots10x_{k-1}\ldots x_0$$

Then the value of X is given by

$$V(X) = -2^{k+1} + x_{k-1} \times 2^{k-1} + \cdots + x_0 \times 2^0$$

The correctness of this expression for $V(X)$ is shown by observing that if X is formed as the sum of two numbers, as follows,

$$
\begin{array}{rl}
 & 11\ldots100000\ldots0 \\
+ & 00\ldots00x_{k-1}\ldots x_0 \\
\hline
X = & 11\ldots10x_{k-1}\ldots x_0
\end{array}
$$

then the upper number is the 2's-complement representation of -2^{k+1}. The recoded multiplier now consists of the part corresponding to the lower number, with -1 added in position $k+1$. For example, the multiplier 110110 is recoded as $0\ -1\ +1\ 0\ -1\ 0$.

The Booth technique for recoding multipliers is summarized in Figure 9.12. The transformation $011\ldots110 \Rightarrow +1\ 0\ 0\ldots0\ -1\ 0$ is called *skipping over 1s*. This term is derived from the case in which the multiplier has its 1s grouped into a few contiguous blocks. Only a few versions of the shifted multiplicand (the summands) need to be added to generate the product, thus speeding up the multiplication operation. However, in the worst case—that of alternating 1s and 0s in the multiplier—each bit of the multiplier selects a summand. In fact, this results in more summands than if the Booth algorithm were not used. A 16-bit worst-case multiplier, an ordinary multiplier, and a good multiplier are shown in Figure 9.13.

The Booth algorithm has two attractive features. First, it handles both positive and negative multipliers uniformly. Second, it achieves some efficiency in the number of additions required when the multiplier has a few large blocks of 1s.

https://hemanthrajhemu.github.io

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i-1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Figure 9.12 Booth multiplier recoding table.

Worst-case multiplier	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	⇓															
	+1	–1	+1	–1	+1	–1	+1	–1	+1	–1	+1	–1	+1	–1	+1	–1
Ordinary multiplier	1	1	0	0	0	1	0	1	1	0	1	1	1	1	0	0
	⇓															
	0	–1	0	0	+1	–1	+1	0	–1	+1	0	0	0	–1	0	0
Good multiplier	0	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1
	⇓															
	0	0	0	+1	0	0	0	0	–1	0	0	0	+1	0	0	–1

Figure 9.13 Booth recoded multipliers.

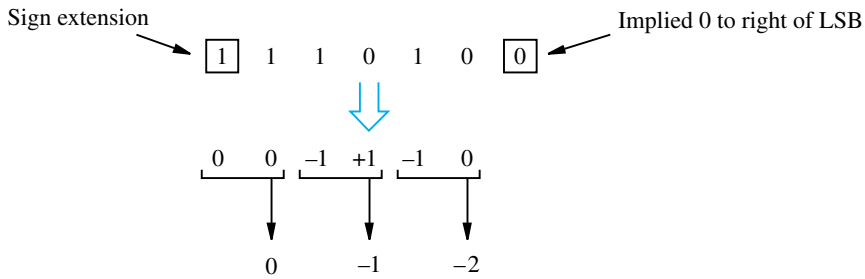
9.5 FAST MULTIPLICATION

We now describe two techniques for speeding up the multiplication operation. The first technique guarantees that the maximum number of summands (versions of the multiplicand) that must be added is $n/2$ for n-bit operands. The second technique leads to adding the summands in parallel.

https://hemanthrajhemu.github.io

9.5.1 BIT-PAIR RECODING OF MULTIPLIERS

A technique called *bit-pair recoding* of the multiplier results in using at most one summand for each pair of bits in the multiplier. It is derived directly from the Booth algorithm. Group the Booth-recoded multiplier bits in pairs, and observe the following. The pair (+1 −1) is equivalent to the pair (0 +1). That is, instead of adding −1 times the multiplicand M at shift position i to +1 × M at position $i + 1$, the same result is obtained by adding +1 × M at position i. Other examples are: (+1 0) is equivalent to (0 +2), (−1 +1) is equivalent to (0 −1), and so on. Thus, if the Booth-recoded multiplier is examined two bits at a time, starting from the right, it can be rewritten in a form that requires at most one version of the multiplicand to be added to the partial product for each pair of multiplier bits. Figure 9.14*a* shows an example of bit-pair recoding of the multiplier in Figure 9.11, and Figure 9.14*b*



(a) Example of bit-pair recoding derived from Booth recoding

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position i
$i + 1$	i	$i - 1$	
0	0	0	0 × M
0	0	1	+ 1 × M
0	1	0	+ 1 × M
0	1	1	+ 2 × M
1	0	0	– 2 × M
1	0	1	– 1 × M
1	1	0	– 1 × M
1	1	1	0 × M

(b) Table of multiplicand selection decisions

Figure 9.14 Multiplier bit-pair recoding.

https://hemanthrajhemu.github.io

$$
\begin{array}{rrrrrrrrrrl}
 & & & & & 0 & 1 & 1 & 0 & 1 & (+13) \\
\times & & & & & 1 & 1 & 0 & 1 & 0 & (-6) \\
\hline
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{rrrrrrrrrrl}
 & & & & & 0 & 1 & 1 & 0 & 1 & \\
 & & & & & 0 & -1 & +1 & -1 & 0 & \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & & \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & & & \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
\hline
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & (-78)
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{rrrrrrrrrrl}
 & & & & & 0 & 1 & 1 & 0 & 1 & \\
 & & & & & 0 & & -1 & & -2 & \\
\hline
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & & & \\
0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
\hline
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 &
\end{array}
$$

Figure 9.15 Multiplication requiring only $n/2$ summands.

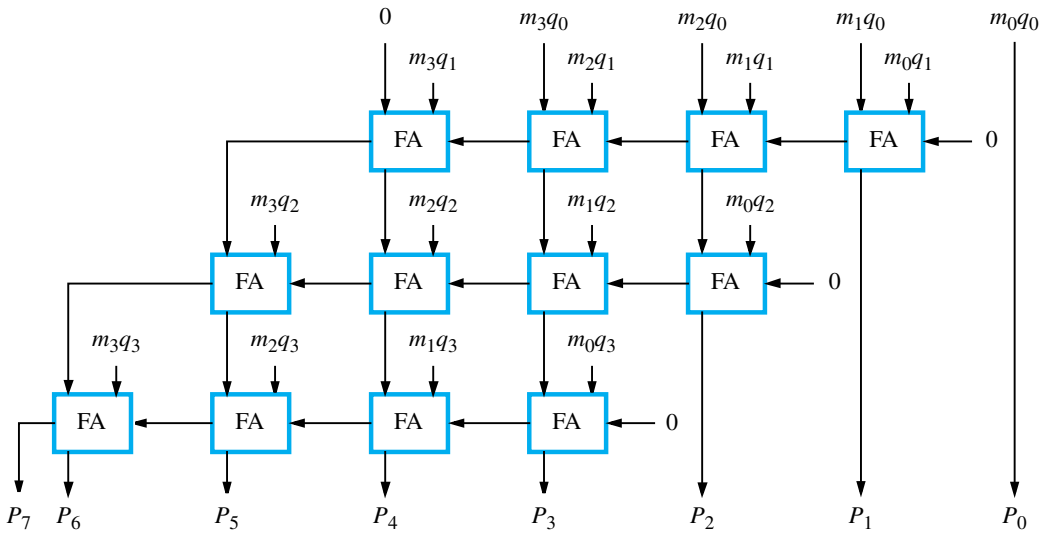
shows a table of the multiplicand selection decisions for all possibilities. The multiplication operation in Figure 9.11 is shown in Figure 9.15 as it would be computed using bit-pair recoding of the multiplier.

9.5.2 CARRY-SAVE ADDITION OF SUMMANDS

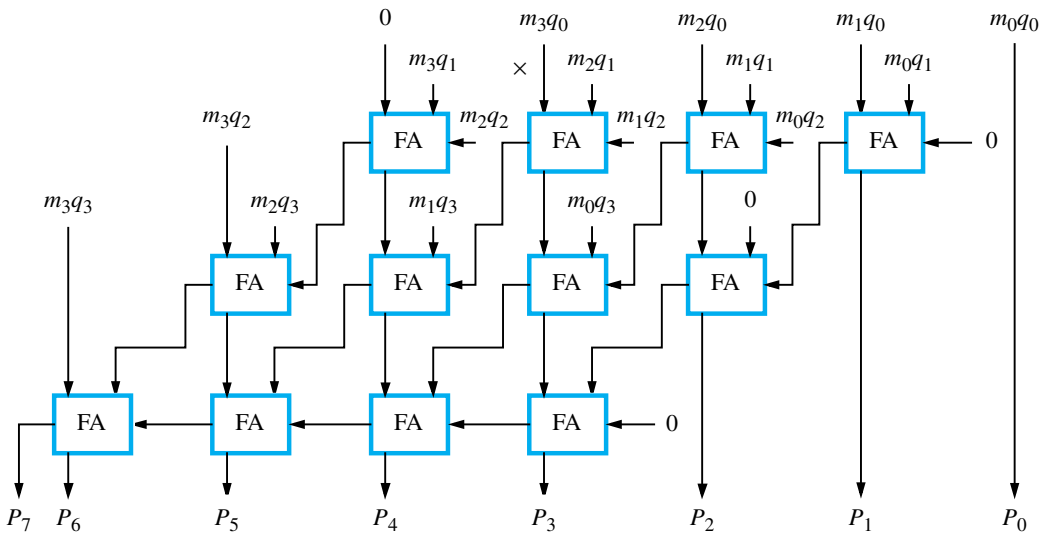
Multiplication requires the addition of several summands. A technique called *carry-save addition* (CSA) can be used to speed up the process. Consider the 4×4 multiplication array shown in Figure 9.16*a*. This structure is in the form of the array shown in Figure 9.6, in which the first row consists of just the AND gates that produce the four inputs m_3q_0, m_2q_0, m_1q_0, and m_0q_0.

Instead of letting the carries ripple along the rows, they can be "saved" and introduced into the next row, at the correct weighted positions, as shown in Figure 9.16*b*. This frees up an input to each of three full adders in the first row. These inputs can be used to introduce

https://hemanthrajhemu.github.io



(a) Ripple-carry array



(b) Carry-save array

Figure 9.16 Ripple-carry and carry-save arrays for a 4 × 4 multiplier.

https://hemanthrajhemu.github.io

the third summand bits m_2q_2, m_1q_2, and m_0q_2. Now, two inputs of each of three full adders in the second row are fed by the sum and carry outputs from the first row. The third input is used to introduce the bits m_2q_3, m_1q_3, and m_0q_3 of the fourth summand. The high-order bits m_3q_2 and m_3q_3 of the third and fourth summands are introduced into the remaining free full-adder inputs at the left end in the second and third rows. The saved carry bits and the sum bits from the second row are now added in the third row, which is a ripple-carry adder, to produce the final product bits.

The delay through the carry-save array is somewhat less than the delay through the ripple-carry array. This is because the S and C vector outputs from each row are produced in parallel in one full-adder delay. The amount of reduction in delay is considered in Problem 9.15.

9.5.3 SUMMAND ADDITION TREE USING 3-2 REDUCERS

A more significant reduction in delay can be achieved when dealing with longer operands than those considered in Figure 9.16. We can group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay. Here, we will refer to a full-adder circuit as simply an adder. Next, we group all the S and C vectors into threes, and perform carry-save addition on them, generating a further set of S and C vectors in one more adder delay. We continue with this process until there are only two vectors remaining. The adder at each bit position of the three summands is called a *3-2 reducer*, and the logic circuit structure that reduces a number of summands to two is called a *CSA tree*, as described by Wallace [2]. The final two S and C vectors can be added in a carry-lookahead adder to produce the desired product.

Consider the example shown in Figure 9.17. It involves adding the six shifted versions of the multiplicand for the case of multiplying two, 6-bit, unsigned numbers, where all six

						1	0	1	1	0	1	(45)	M
					×	1	1	1	1	1	1	(63)	Q
						1	0	1	1	0	1	A	
					1	0	1	1	0	1		B	
				1	0	1	1	0	1			C	
			1	0	1	1	0	1				D	
		1	0	1	1	0	1					E	
	1	0	1	1	0	1						F	
1	0	1	1	0	0	0	1	0	0	1	1	(2,835)	Product

Figure 9.17 A multiplication example used to illustrate carry-save addition as shown in Figure 9.18.

https://hemanthrajhemu.github.io

bits of the multiplier are equal to 1. The six summands, $A, B, \ldots, F$ are added by carry-save addition in Figure 9.18. The blue boxes in these two figures indicate the same operand bits, and show how they are reduced to sum and carry bits in Figure 9.18 by carry-save addition. Three levels of carry-save addition are performed, as shown schematically in Figure 9.19. This figure shows that the final two vectors S_4 and C_4 are available in three adder delays

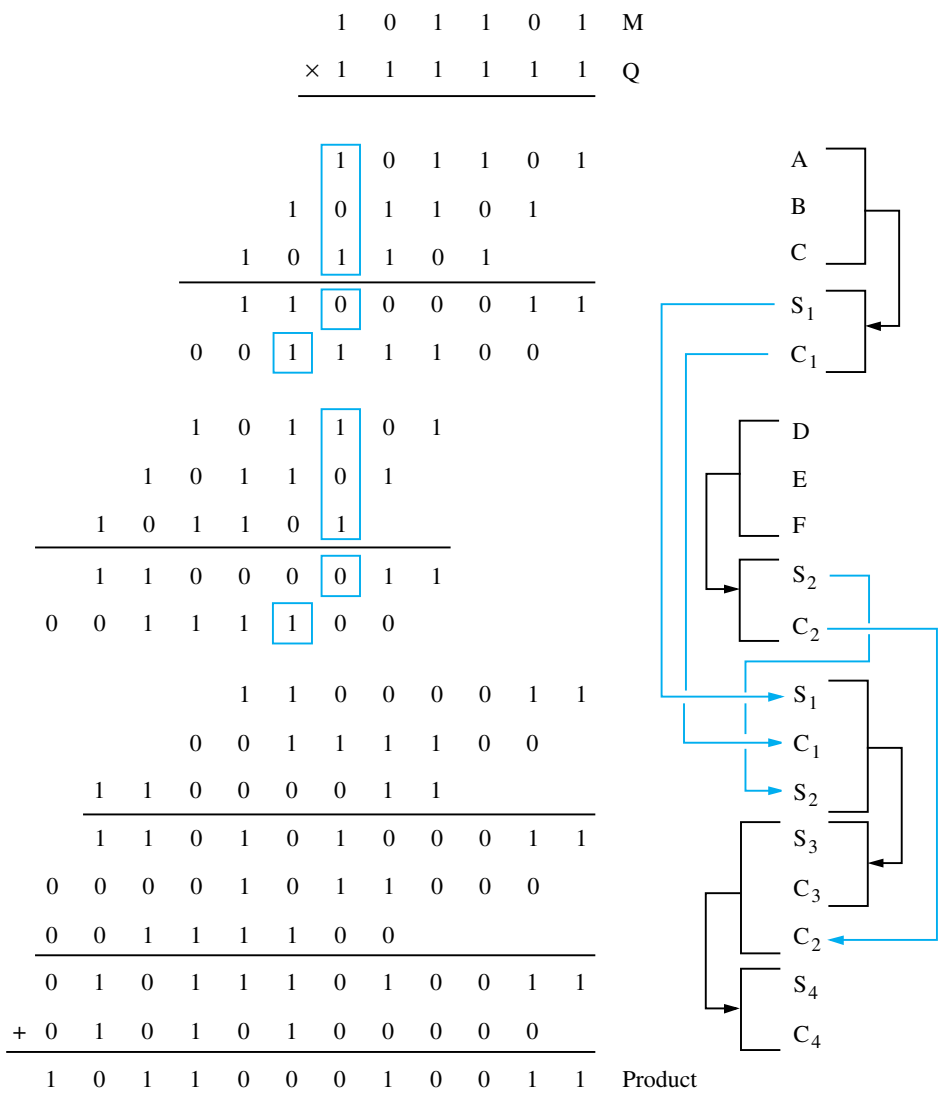


Figure 9.18 The multiplication example from Figure 9.17 performed using carry-save addition.

https://hemanthrajhemu.github.io

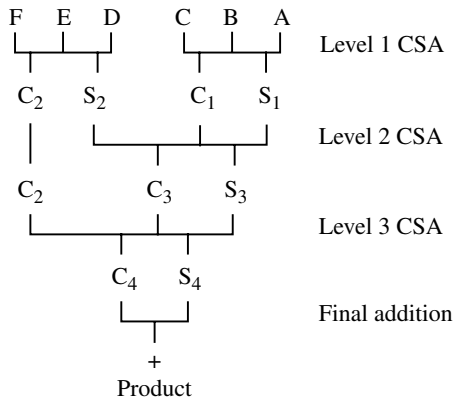


Figure 9.19 Schematic representation of the carry-save addition operations in Figure 9.18.

after the six input summands are applied to level 1. The final regular addition operation on S_4 and C_4, which produces the product, can be done with a carry-lookahead adder.

The multiplier delay is lower when using the tree structure illustrated in Figure 9.19 than when using the array structure illustrated in Figure 9.16*b*. When the number of summands is large, the reduction in delay is significant. For example, the addition of 32 summands following the pattern shown in Figure 9.19 requires only 8 levels of 3-2 reduction before the final Add operation. In general, it can be shown that approximately $1.7log_2k - 1.7$ levels of 3-2 reduction are needed to reduce k summands to 2 vectors, which, when added, produce the desired product. (See Example 9.3. in Section 9.10.)

We should note that negative summands are involved when signed-number multiplication and Booth recoding of multipliers is used. This requires sign extension of the summands before they are entered into the reduction tree. Also, the number of summands that need to be added is reduced if bit-pair recoding of the multiplier is done.

The 3-2 reducer is not the only logic circuit that can be used in building reduction trees. It is also possible to use 4-2 reducers and 7-3 reducers. The first of these possibilities is described in the next subsection, and the second is explored in Problem 9.17.

9.5.4 SUMMAND ADDITION TREE USING 4-2 REDUCERS

The interconnection pattern between levels in a CSA tree that uses 3-2 reducers is irregular, as can be seen in Figure 9.19. A more regularly structured tree can be obtained by using 4-2 reducers [3], especially for the case in which the number of summands to be reduced is a power of 2. This is the usual case for the multiplication operation in the ALU of a processor. For example, if 32 summands are reduced to 2 using 4-2 reducers at each reduction level, then only four levels are needed. The tree has a regular structure, with 16, 8, 4, and 2 summands at the outputs of the four levels. If 3-2 reducers are used, eight levels

https://hemanthrajhemu.github.io

are required, and the wiring connections between levels are quite irregular. Regular tree structures facilitate logic circuit and wiring layout for VLSI circuit implementation.

Let us consider the design of a 4-2 reducer as developed in reference [3]. The addition of four equally-weighted bits, w, x, y, and z, from four summands, produces a value in the range 0 to 4. Such a value cannot be represented by a sum bit, s, and a single carry bit, c. However, a second carry bit, c_{out}, with the same weight as c, can be used along with s and c, to represent any value in the range 0 to 5. This is sufficient for our purposes here.

We do not want to send three output bits down to the next reduction level. That would implement a 4-3 reducer, which provides less reduction than a 3-2 reducer. The solution is to send c_{out} laterally to the 4-2 reducer in the next higher-weighted bit position on the same reduction level. Thus, each 4-2 reducer must have a fifth input, c_{in}, which is the c_{out} output from the 4-2 reducer in the next lower-weighted bit position on the same reduction level.

A final requirement on the design of the 4-2 reducer is that the value of c_{out} cannot depend on the value of c_{in}. This is a key requirement. Without it, carries would ripple laterally along a reduction level, defeating the purpose of parallel reduction of summands with short fixed delay. A 4-2 reducer block is shown in Figure 9.20.

In summary, the specification for a 4-2 reducer is as follows:

- The three outputs, s, c, and c_{out}, represent the arithmetic sum of the five inputs, that is

$$w + x + y + z + c_{in} = s + 2(c + c_{out})$$

  where all operators here are arithmetic.

- Output s is the usual sum variable; that is, s is the XOR function of the five input variables.

- The lateral carry, c_{out}, must be independent of c_{in}. It is a function of only the four input variables w, x, y, and z.

There are different possibilities for specifying the two carry outputs in a way that meets the given conditions. We present one that is easy to describe. First, assign the lateral carry

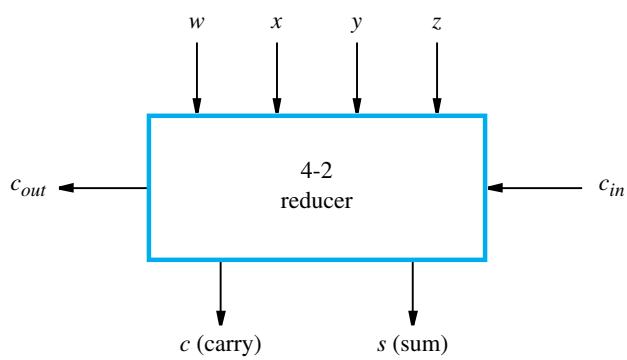


Figure 9.20 A 4-2 reducer block.

https://hemanthrajhemu.github.io

				$c_{in} = 0$		$c_{in} = 1$		
w	x	y	z	c	s	c	s	c_{out}
0	0	0	0	0	0	0	1	0
0	0	0	1	0	1	1	0	0
0	0	1	0	0	1	1	0	0
0	1	0	0	0	1	1	0	0
1	0	0	0	0	1	1	0	0
0	0	1	1	0	0	0	1	1
0	1	0	1	0	0	0	1	1
0	1	1	0	0	0	0	1	1
1	0	0	1	0	0	0	1	1
1	0	1	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
0	1	1	1	0	1	1	0	1
1	0	1	1	0	1	1	0	1
1	1	0	1	0	1	1	0	1
1	1	1	0	0	1	1	0	1
1	1	1	1	1	0	1	1	1

Figure 9.21 A 4-2 reducer truth table.

output, c_{out}, to be 1 when two or more of the input variables w, x, y, and z, are equal to 1. Then, the other carry output, c, is determined so as to satisfy the arithmetic condition. A complete truth table satisfying these conditions is given in Figure 9.21. The table is shown in a form that is different from the usual form used in Appendix A. The four inputs w, x, y, and z, are not listed in binary numerical order. They are listed in groups corresponding to the number of inputs that have the value 1. This makes it easy to see how the outputs are specified to meet the given conditions. A logic gate network can be derived from the table.

9.5.5 SUMMARY OF FAST MULTIPLICATION

We now summarize the techniques for high-speed multiplication. Bit-pair recoding of the multiplier, derived from the Booth algorithm, can be used to initially reduce the number of summands by a factor of two. The resulting summands can then be reduced to two in a reduction tree with a relatively small number of reduction levels. The final product

https://hemanthrajhemu.github.io

can be generated by an addition operation that uses a carry-lookahead adder. All three of these techniques—bit-pair recoding of the multiplier, parallel reduction of summands, and carry-lookahead addition—have been used in various combinations by the designers of high-performance processors to reduce the time needed to perform multiplication.

9.6 Integer Division

In Section 9.3, we discussed the multiplication of unsigned numbers by relating the way the multiplication operation is done manually to the way it is done in a logic circuit. We use the same approach here in discussing integer division. We discuss unsigned-number division in detail, and then make some general comments on the signed-number case.

Figure 9.22 shows examples of decimal division and binary division of the same values. Consider the decimal version first. The 2 in the quotient is determined by the following reasoning: First, we try to divide 13 into 2, and it does not work. Next, we try to divide 13 into 27. We go through the trial exercise of multiplying 13 by 2 to get 26, and, observing that $27 - 26 = 1$ is less than 13, we enter 2 as the quotient and perform the required subtraction. The next digit of the dividend, 4, is brought down, and we finish by deciding that 13 goes into 14 once, and the remainder is 1. We can discuss binary division in a similar way, with the simplification that the only possibilities for the quotient bits are 0 and 1.

A circuit that implements division by this longhand method operates as follows: It positions the divisor appropriately with respect to the dividend and performs a subtraction. If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed. If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction. This is called the *restoring division* algorithm.

Restoring Division

Figure 9.23 shows a logic circuit arrangement that implements the restoring division algorithm just discussed. Note its similarity to the structure for multiplication shown in Figure 9.7. An n-bit positive divisor is loaded into register M and an n-bit positive dividend

```
      21                      10101
13 ) 274            1101 ) 100010010
     26                     1101
     --                     -----
      14                     10000
      13                      1101
      --                     -----
       1                       1110
                               1101
                               ----
                                  1
```

Figure 9.22 Longhand division examples.

https://hemanthrajhemu.github.io

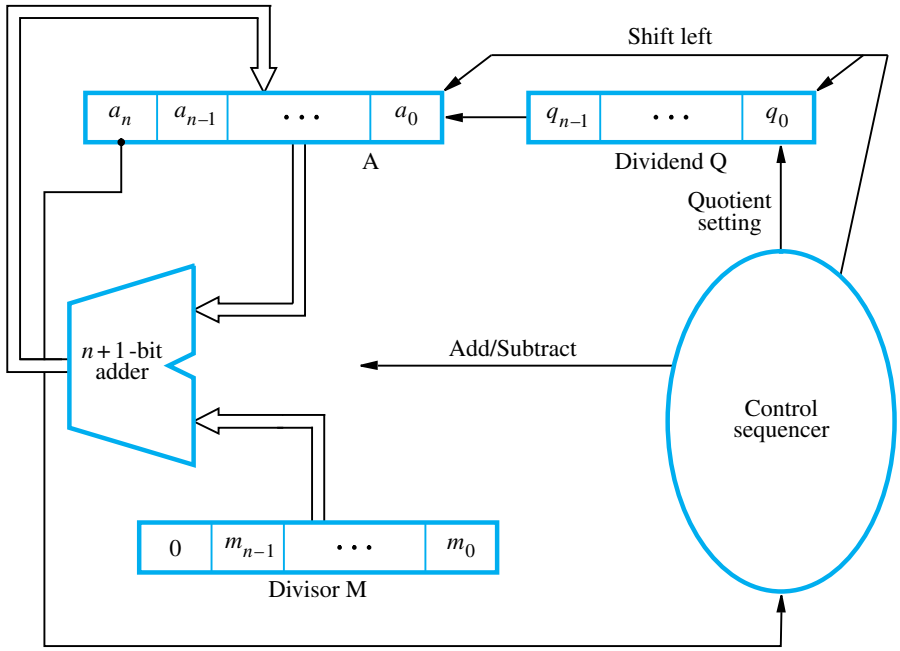


Figure 9.23 Circuit arrangement for binary division.

is loaded into register Q at the start of the operation. Register A is set to 0. After the division is complete, the n-bit quotient is in register Q and the remainder is in register A. The required subtractions are facilitated by using 2's-complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions. The following algorithm performs restoring division.

Do the following three steps n times:

1. Shift A and Q left one bit position.
2. Subtract M from A, and place the answer back in A.
3. If the sign of A is 1, set q_0 to 0 and add M back to A (that is, restore A); otherwise, set q_0 to 1.

Figure 9.24 shows a 4-bit example as it would be processed by the circuit in Figure 9.23.

Non-Restoring Division

The restoring division algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction. Subtraction is said to be unsuccessful if the result is negative. Consider the sequence of operations that takes place after the subtraction operation in the preceding algorithm. If A is positive, we shift left and subtract M, that is, we perform $2A - M$. If A is negative, we restore it by performing $A + M$, and then we shift it left and subtract M. This is equivalent to performing $2A + M$. The q_0 bit is appropriately

https://hemanthrajhemu.github.io

```
        10
11 ) 1000
      11
      10
```

	Remainder	Quotient	
Initially	0 0 0 0 0	1 0 0 0	First cycle
	0 0 0 1 1		
Shift	0 0 0 0 1	0 0 0 □	
Subtract	1 1 1 0 1		
Set q_0	1 1 1 1 0		
Restore	1 1		
	0 0 0 0 1	0 0 0 0	
Shift	0 0 0 1 0	0 0 0 □	Second cycle
Subtract	1 1 1 0 1		
Set q_0	1 1 1 1 1		
Restore	1 1		
	0 0 0 1 0	0 0 0 0	
Shift	0 0 1 0 0	0 0 0 □	Third cycle
Subtract	1 1 1 0 1		
Set q_0	0 0 0 0 1		
Shift	0 0 0 1 0	0 0 0 1	
Subtract	1 1 1 0 1	0 0 1 □	Fourth cycle
Set q_0	1 1 1 1 1		
Restore	1 1		
	0 0 0 1 0	0 0 1 0	

Figure 9.24 A restoring division example.

set to 0 or 1 after the correct operation has been performed. We can summarize this in the following algorithm for *non-restoring division*.

Stage 1: Do the following two steps n times:

1. If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
2. Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.

Stage 2: If the sign of A is 1, add M to A.

Stage 2 is needed to leave the proper positive remainder in A after the n cycles of Stage 1. The logic circuitry in Figure 9.23 can also be used to perform this algorithm, except that

https://hemanthrajhemu.github.io

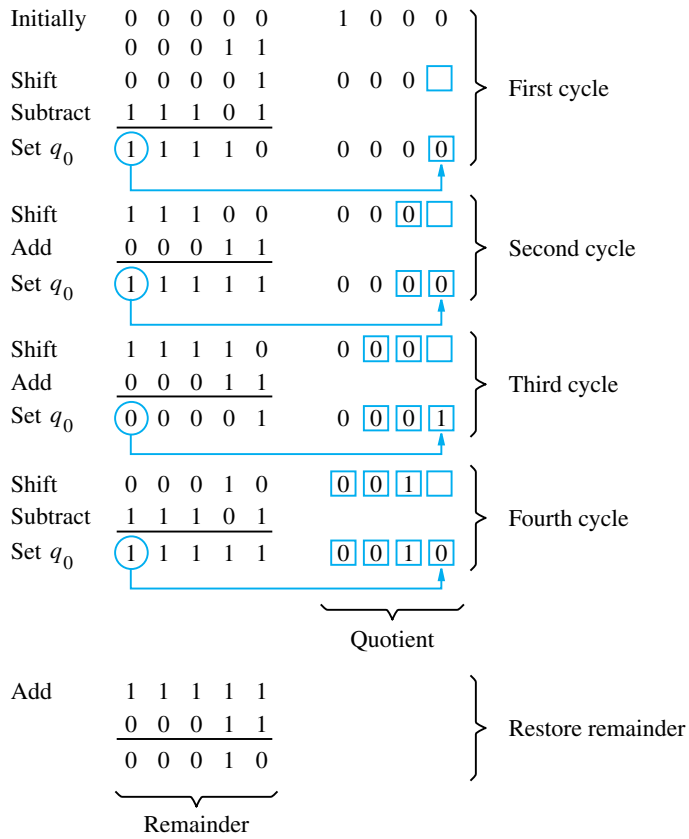


Figure 9.25 A non-restoring division example.

the Restore operations are no longer needed. One Add or Subtract operation is performed in each of the n cycles of stage 1, plus a possible final addition in Stage 2. Figure 9.25 shows how the division example in Figure 9.24 is executed by the non-restoring division algorithm.

There are no simple algorithms for directly performing division on signed operands that are comparable to the algorithms for signed multiplication. In division, the operands can be preprocessed to change them into positive values. After using one of the algorithms just discussed, the signs of the quotient and the remainder are adjusted as necessary.

9.7 FLOATING-POINT NUMBERS AND OPERATIONS

Chapter 1 provided the motivation for using floating-point numbers and indicated how they can be represented in a 32-bit binary format. In this chapter, we provide more detail on representation formats and arithmetic operations on floating-point numbers. The descriptions

https://hemanthrajhemu.github.io

provided here are based on the 2008 version of IEEE (Institute of Electrical and Electronics Engineers) Standard 754, labeled 754-2008 [4].

Recall from Chapter 1 that a binary floating-point number can be represented by

- A sign for the number
- Some significant bits
- A signed scale factor exponent for an implied base of 2

The basic IEEE format is a 32-bit representation, shown in Figure 9.26*a*. The leftmost bit represents the sign, S, for the number. The next 8 bits, E', represent the signed exponent of the scale factor (with an implied base of 2), and the remaining 23 bits, M, are the

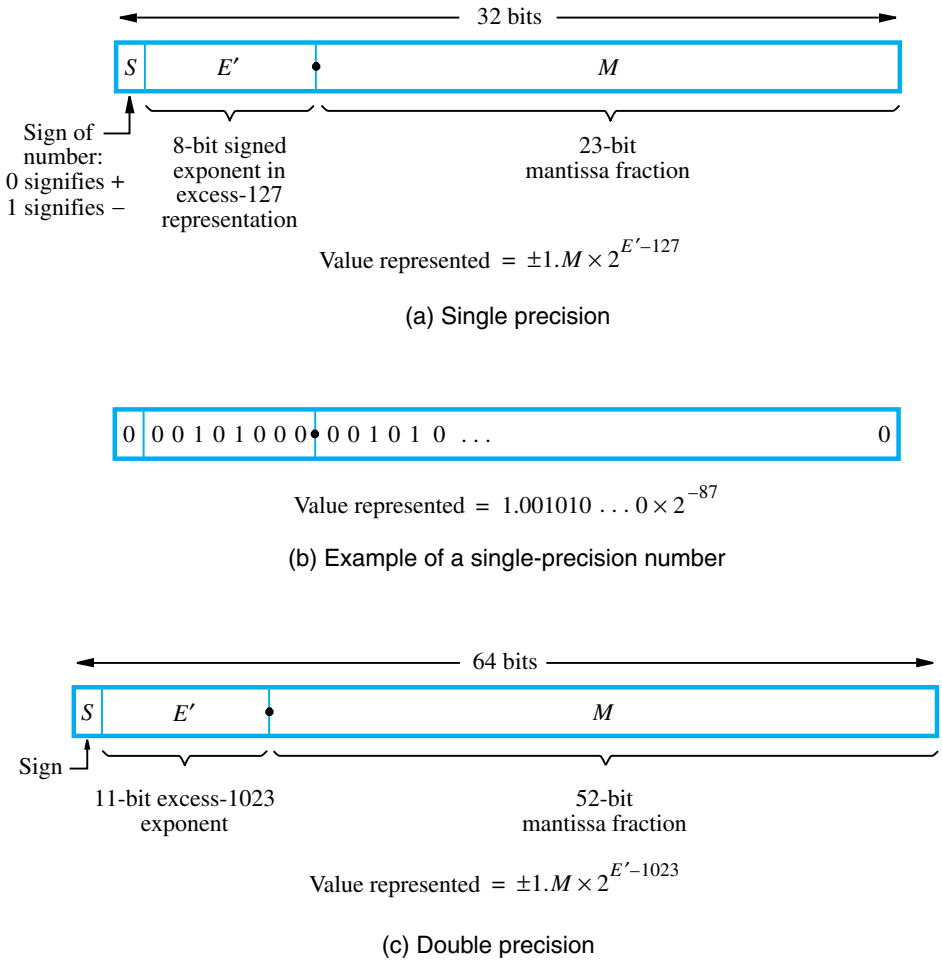


Figure 9.26 IEEE standard floating-point formats.

https://hemanthrajhemu.github.io

fractional part of the significant bits. The full 24-bit string, B, of significant bits, called the *mantissa*, always has a leading 1, with the binary point immediately to its right. Therefore, the mantissa

$$B = 1.M = 1.b_{-1}b_{-2}\ldots b_{-23}$$

has the value

$$V(B) = 1 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-23} \times 2^{-23}$$

By convention, when the binary point is placed to the right of the first significant bit, the number is said to be *normalized*. Note that the base, 2, of the scale factor and the leading 1 of the mantissa are both fixed. They do not need to appear explicitly in the representation.

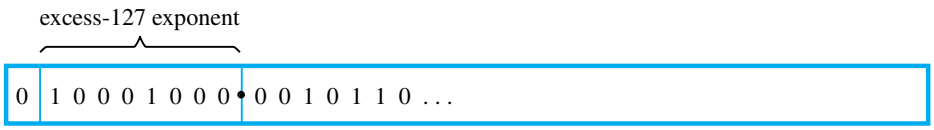
Instead of the actual signed exponent, E, the value stored in the exponent field is an unsigned integer $E' = E + 127$. This is called the *excess*-127 format. Thus, E' is in the range $0 \leq E' \leq 255$. The end values of this range, 0 and 255, are used to represent special values, as described later. Therefore, the range of E' for normal values is $1 \leq E' \leq 254$. This means that the actual exponent, E, is in the range $-126 \leq E \leq 127$. The use of the excess-127 representation for exponents simplifies comparison of the relative sizes of two floating-point numbers. (See Problem 9.23.)

The 32-bit standard representation in Figure 9.26*a* is called a *single-precision* representation because it occupies a single 32-bit word. The scale factor has a range of 2^{-126} to 2^{+127}, which is approximately equal to $10^{\pm 38}$. The 24-bit mantissa provides approximately the same precision as a 7-digit decimal value. An example of a single-precision floating-point number is shown in Figure 9.26*b*.

To provide more precision and range for floating-point numbers, the IEEE standard also specifies a *double-precision* format, as shown in Figure 9.26*c*. The double-precision format has increased exponent and mantissa ranges. The 11-bit excess-1023 exponent E' has the range $1 \leq E' \leq 2046$ for normal values, with 0 and 2047 used to indicate special values, as before. Thus, the actual exponent E is in the range $-1022 \leq E \leq 1023$, providing scale factors of 2^{-1022} to 2^{1023} (approximately $10^{\pm 308}$). The 53-bit mantissa provides a precision equivalent to about 16 decimal digits.

A computer must provide at least single-precision representation to conform to the IEEE standard. Double-precision representation is optional. The standard also specifies certain optional extended versions of both of these formats. The extended versions provide increased precision and increased exponent range for the representation of intermediate values in a sequence of calculations. The use of extended formats helps to reduce the size of the accumulated round-off error in a sequence of calculations leading to a desired result. For example, the dot product of two vectors of numbers involves accumulating a sum of products. The input vector components are given in a standard precision, either single or double, and the final answer (the dot product) is truncated to the same precision. All intermediate calculations should be done using extended precision to limit accumulation of errors. Extended formats also enhance the accuracy of evaluation of elementary functions such as sine, cosine, and so on. This is because they are usually evaluated by adding up a number of terms in a series representation. In addition to requiring the four basic arithmetic operations, the standard requires three additional operations to be provided: remainder, square root, and conversion between binary and decimal representations.

https://hemanthrajhemu.github.io



(There is no implicit 1 to the left of the binary point.)

$$\text{Value represented} = +0.0010110\ldots \times 2^9$$

(a) Unnormalized value

$$\text{Value represented} = +1.0110\ldots \times 2^6$$

(b) Normalized version

Figure 9.27 Floating-point normalization in IEEE single-precision format.

We note two basic aspects of operating with floating-point numbers. First, if a number is not normalized, it can be put in normalized form by shifting the binary point and adjusting the exponent. Figure 9.27 shows an unnormalized value, $0.0010110\ldots \times 2^9$, and its normalized version, $1.0110\ldots \times 2^6$. Since the scale factor is in the form 2^i, shifting the mantissa right or left by one bit position is compensated by an increase or a decrease of 1 in the exponent, respectively. Second, as computations proceed, a number that does not fall in the representable range of normal numbers might be generated. In single precision, this means that its normalized representation requires an exponent less than -126 or greater than $+127$. In the first case, we say that *underflow* has occurred, and in the second case, we say that *overflow* has occurred.

Special Values

The end values 0 and 255 of the excess-127 exponent E' are used to represent special values. When $E' = 0$ and the mantissa fraction M is zero, the value 0 is represented. When $E' = 255$ and $M = 0$, the value ∞ is represented, where ∞ is the result of dividing a normal number by zero. The sign bit is still used in these representations, so there are representations for ± 0 and $\pm\infty$.

When $E' = 0$ and $M \neq 0$, *denormal* numbers are represented. Their value is $\pm 0.M \times 2^{-126}$. Therefore, they are smaller than the smallest normal number. There is no implied one to the left of the binary point, and M is any nonzero 23-bit fraction. The purpose of introducing denormal numbers is to allow for *gradual underflow*, providing an extension of the range of normal representable numbers. This is useful in dealing with very small numbers, which may be needed in certain situations. When $E' = 255$ and $M \neq 0$, the value

https://hemanthrajhemu.github.io

represented is called *Not a Number* (NaN). A NaN represents the result of performing an invalid operation such as $0/0$ or $\sqrt{-1}$.

Exceptions

In conforming to the IEEE Standard, a processor must set *exception* flags if any of the following conditions arise when performing operations: underflow, overflow, divide by zero, inexact, invalid. We have already mentioned the first three. *Inexact* is the name for a result that requires rounding in order to be represented in one of the normal formats. An *invalid* exception occurs if operations such as $0/0$ or $\sqrt{-1}$ are attempted. When an exception occurs, the result is set to one of the special values.

If interrupts are enabled for any of the exception flags, system or user-defined routines are entered when the associated exception occurs. Alternatively, the application program can test for the occurrence of exceptions, as necessary, and decide how to proceed.

9.7.1 ARITHMETIC OPERATIONS ON FLOATING-POINT NUMBERS

In this section, we outline the general procedures for addition, subtraction, multiplication, and division of floating-point numbers. The rules given below apply to the single-precision IEEE standard format. These rules specify only the major steps needed to perform the four operations; for example, the possibility that overflow or underflow might occur is not discussed. Furthermore, intermediate results for both mantissas and exponents might require more than 24 and 8 bits, respectively. These and other aspects of the operations must be carefully considered in designing an arithmetic unit that meets the standard. Although we do not provide full details in specifying the rules, we consider some aspects of implementation, including rounding, in later sections.

When adding or subtracting floating-point numbers, their mantissas must be shifted with respect to each other if their exponents differ. Consider a decimal example in which we wish to add 2.9400×10^2 to 4.3100×10^4. We rewrite 2.9400×10^2 as 0.0294×10^4 and then perform addition of the mantissas to get 4.3394×10^4. The rule for addition and subtraction can be stated as follows:

Add/Subtract Rule

1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2. Set the exponent of the result equal to the larger exponent.
3. Perform addition/subtraction on the mantissas and determine the sign of the result.
4. Normalize the resulting value, if necessary.

Multiplication and division are somewhat easier than addition and subtraction, in that no alignment of mantissas is needed.

Multiply Rule

1. Add the exponents and subtract 127 to maintain the excess-127 representation.
2. Multiply the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

https://hemanthrajhemu.github.io

Divide Rule

1. Subtract the exponents and add 127 to maintain the excess-127 representation.
2. Divide the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

9.7.2 Guard Bits and Truncation

Let us consider some important aspects of implementing the steps in the preceding algorithms. Although the mantissas of initial operands and final results are limited to 24 bits, including the implicit leading 1, it is important to retain extra bits, often called *guard* bits, during the intermediate steps. This yields maximum accuracy in the final results.

Removing guard bits in generating a final result requires that the extended mantissa be *truncated* to create a 24-bit number that approximates the longer version. This operation also arises in other situations, for instance, in converting from decimal to binary numbers. We should mention that the general term rounding is also used for the truncation operation, but a more restrictive definition of rounding is used here as one of the forms of truncation.

There are several ways to truncate. The simplest way is to remove the guard bits and make no changes in the retained bits. This is called *chopping*. Suppose we want to truncate a fraction from six to three bits by this method. All fractions in the range $0.b_{-1}b_{-2}b_{-3}000$ to $0.b_{-1}b_{-2}b_{-3}111$ are truncated to $0.b_{-1}b_{-2}b_{-3}$. The error in the 3-bit result ranges from 0 to 0.000111. In other words, the error in chopping ranges from 0 to almost 1 in the least significant position of the retained bits. In our example, this is the b_{-3} position. The result of chopping is a *biased* approximation because the error range is not symmetrical about 0.

The next simplest method of truncation is *von Neumann rounding*. If the bits to be removed are all 0s, they are simply dropped, with no changes to the retained bits. However, if any of the bits to be removed are 1, the least significant bit of the retained bits is set to 1. In our 6-bit to 3-bit truncation example, all 6-bit fractions with $b_{-4}b_{-5}b_{-6}$ not equal to 000 are truncated to $0.b_{-1}b_{-2}1$. The error in this truncation method ranges between -1 and $+1$ in the LSB position of the retained bits. Although the range of error is larger with this technique than it is with chopping, the maximum magnitude is the same, and the approximation is *unbiased* because the error range is symmetrical about 0.

Unbiased approximations are advantageous if many operands and operations are involved in generating a result, because positive errors tend to offset negative errors as the computation proceeds. Statistically, we can expect the results of a complex computation to be more accurate.

The third truncation method is a *rounding* procedure. Rounding achieves the closest approximation to the number being truncated and is an unbiased technique. The procedure is as follows: A 1 is added to the LSB position of the bits to be retained if there is a 1 in the MSB position of the bits being removed. Thus, $0.b_{-1}b_{-2}b_{-3}1\ldots$ is rounded to $0.b_{-1}b_{-2}b_{-3} + 0.001$, and $0.b_{-1}b_{-2}b_{-3}0\ldots$ is rounded to $0.b_{-1}b_{-2}b_{-3}$. This provides the desired approximation, except for the case in which the bits to be removed are $10\ldots0$. This is a tie situation; the longer value is halfway between the two closest truncated representations. To break the tie in an unbiased way, one possibility is to choose the retained

https://hemanthrajhemu.github.io

bits to be the nearest even number. In terms of our 6-bit example, the value $0.b_{-1}b_{-2}0100$ is truncated to the value $0.b_{-1}b_{-2}0$, and $0.b_{-1}b_{-2}1100$ is truncated to $0.b_{-1}b_{-2}1 + 0.001$. The descriptive phrase "round to the nearest number or nearest even number in case of a tie" is sometimes used to refer to this truncation technique. The error range is approximately $-\frac{1}{2}$ to $+\frac{1}{2}$ in the LSB position of the retained bits. Clearly, this is the best method. However, it is also the most difficult to implement because it requires an addition operation and a possible renormalization. This rounding technique is the default mode for truncation specified in the IEEE floating-point standard. The standard also specifies other truncation methods, referring to all of them as rounding modes.

This discussion of errors that are introduced when guard bits are removed by truncation has treated the case of a single truncation operation. When a long series of calculations involving floating-point numbers is performed, the analysis that determines error ranges or bounds for the final results can be a complicated study. We do not discuss this aspect of numerical computation further, except to make a few comments on the way that guard bits and rounding are handled in the IEEE floating-point standard.

According to the standard, results of single operations must be computed to be accurate within half a unit in the LSB position. This means that rounding must be used as the truncation method. Implementing rounding requires only three guard bits to be carried along during the intermediate steps in performing an operation. The first two of these bits are the two most significant bits of the section of the mantissa to be removed. The third bit is the logical OR of all bits beyond these first two bits in the full representation of the mantissa. This bit is relatively easy to maintain during the intermediate steps of the operations to be performed. It should be initialized to 0. If a 1 is shifted out through this position while aligning mantissas, the bit becomes 1 and retains that value; hence, it is usually called the *sticky bit*.

9.7.3 IMPLEMENTING FLOATING-POINT OPERATIONS

The hardware implementation of floating-point operations involves a considerable amount of logic circuitry. These operations can also be implemented by software routines. In either case, the computer must be able to convert input and output from and to the user's decimal representation of numbers. In many general-purpose processors, floating-point operations are available at the machine-instruction level, implemented in hardware.

An example of the implementation of floating-point operations is shown in Figure 9.28. This is a block diagram of a hardware implementation for the addition and subtraction of 32-bit floating-point operands that have the format shown in Figure 9.26*a*. Following the Add/Subtract rule given in Section 9.7.1, we see that the first step is to compare exponents to determine how far to shift the mantissa of the number with the smaller exponent. The shift-count value, n, is determined by the 8-bit subtractor circuit in the upper left corner of the figure. The magnitude of the difference $E'_A - E'_B$, or n, is sent to the SHIFTER unit. If n is larger than the number of significant bits of the operands, then the answer is essentially the larger operand (except for guard and sticky-bit considerations in rounding), and shortcuts can be taken in deriving the result. We do not explore this in detail.

https://hemanthrajhemu.github.io

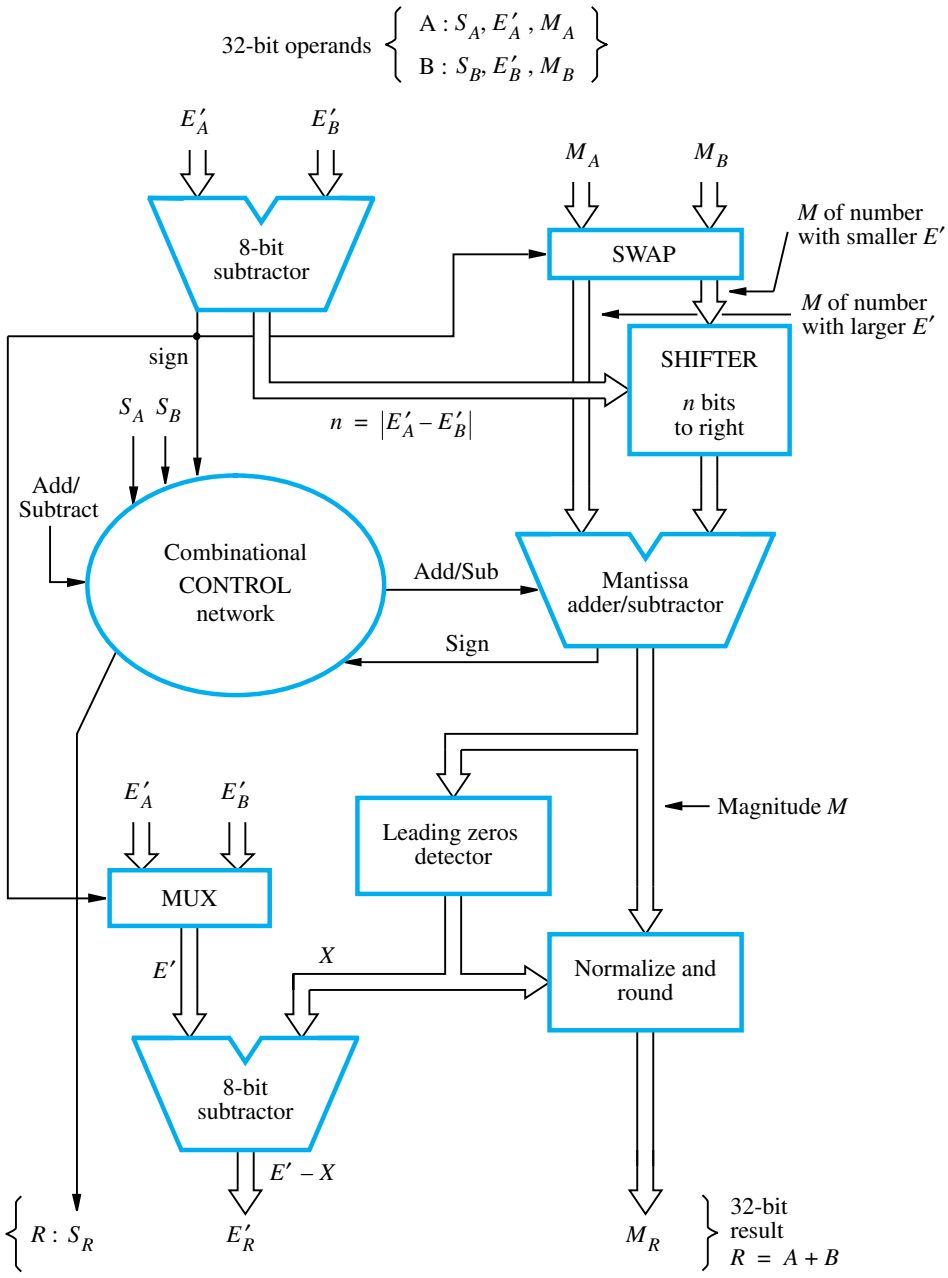


Figure 9.28 Floating-point addition-subtraction unit.

https://hemanthrajhemu.github.io

The sign of the difference that results from comparing exponents determines which mantissa is to be shifted. Therefore, in step 1, the sign is sent to the SWAP network in the upper right corner of Figure 9.28. If the sign is 0, then $E'_A \geq E'_B$ and the mantissas M_A and M_B are sent straight through the SWAP network. This results in M_B being sent to the SHIFTER, to be shifted n positions to the right. The other mantissa, M_A, is sent directly to the mantissa adder/subtractor. If the sign is 1, then $E'_A < E'_B$ and the mantissas are swapped before they are sent to the SHIFTER.

Step 2 is performed by the two-way multiplexer, MUX, near the bottom left corner of the figure. The exponent of the result, E', is tentatively determined as E'_A if $E'_A \geq E'_B$, or E'_B if $E'_A < E'_B$, based on the sign of the difference resulting from comparing exponents in step 1.

Step 3 involves the major component, the mantissa adder/subtractor in the middle of the figure. The CONTROL logic determines whether the mantissas are to be added or subtracted. This is decided by the signs of the operands (S_A and S_B) and the operation (Add or Subtract) that is to be performed on the operands. The CONTROL logic also determines the sign of the result, S_R. For example, if A is negative ($S_A = 1$), B is positive ($S_B = 0$), and the operation is $A - B$, then the mantissas are added and the sign of the result is negative ($S_R = 1$). On the other hand, if A and B are both positive and the operation is $A - B$, then the mantissas are subtracted. The sign of the result, S_R, now depends on the mantissa subtraction operation. For instance, if $E'_A > E'_B$, then $M = M_A -$ (shifted M_B) and the resulting number is positive. But if $E'_B > E'_A$, then $M = M_B -$ (shifted M_A) and the result is negative. This example shows that the sign from the exponent comparison is also required as an input to the CONTROL network. When $E'_A = E'_B$ and the mantissas are subtracted, the sign of the mantissa adder/subtractor output determines the sign of the result. The reader should now be able to construct the complete truth table for the CONTROL network (see Problem 9.26).

Step 4 of the Add/Subtract rule consists of normalizing the result of step 3 by shifting M to the right or to the left, as appropriate. The number of leading zeros in M determines the number of bit shifts, X, to be applied to M. The normalized value is rounded to generate the 24-bit mantissa, M_R, of the result. The value X is also subtracted from the tentative result exponent E' to generate the true result exponent, E'_R. Note that only a single right shift might be needed to normalize the result. This would be the case if two mantissas of the form $1.xx\ldots$ were added. The vector M would then have the form $1x.xx\ldots$.

We have not given any details on the guard bits that must be carried along with intermediate mantissa values. In the IEEE standard, only a few bits are needed, as discussed earlier, to generate the 24-bit normalized mantissa of the result.

Let us consider the actual hardware that is needed to implement the blocks in Figure 9.28. The two 8-bit subtractors and the mantissa adder/subtractor can be implemented by combinational logic, as discussed earlier in this chapter. Because their outputs must be in sign-and-magnitude form, we must modify some of our earlier discussions. A combination of 1's-complement arithmetic and sign-and-magnitude representation is often used. Considerable flexibility is allowed in implementing the SHIFTER and the output normalization operation. The operations can be implemented with shift registers. However, they can also be built as combinational logic units for high-performance.

https://hemanthrajhemu.github.io

9.8 Decimal-to-Binary Conversion

In Chapter 1 and in this chapter, examples that involve decimal numbers have used small values. Conversion from decimal to binary representation has been easy to do based on the binary bit-position weights 1, 2, 4, 8, 16, ... However, it is useful to have a general method for converting decimal numbers to binary representation.

The fixed-point, unsigned, binary number

$$B = b_{n-1}b_{n-2}\ldots b_0.b_{-1}b_{-2}\ldots b_{-m}$$

has an n-bit integer part and an m-bit fraction part. Its value, $V(B)$, is given by

$$\begin{aligned} V(B) = {} & b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_0 \times 2^0 \\ & + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-m} \times 2^{-m} \end{aligned}$$

To convert a fixed-point decimal number into binary, the integer and fraction parts are handled separately. Conversion of the integer part starts by dividing it by 2. The remainder, which is either 0 or 1, is the least significant bit, b_0, of the integer part of B. The quotient is again divided by 2. The remainder is the next bit, b_1, of B. This process is repeated up to and including the step in which the quotient becomes 0.

Conversion of the fraction part starts by multiplying it by 2. The part of the product to the left of the decimal point, which is either 0 or 1, is bit b_{-1} of the fraction part of B. The fraction part of the product is again multiplied by 2, generating the next bit, b_{-2} of the fraction part of B. The process is repeated until the fraction part of the product becomes 0 or until the required accuracy is obtained.

Figure 9.29 shows an example of conversion from the decimal number 927.45 to binary. Note that conversion of the integer part is always exact and terminates when the quotient becomes 0. But an exact binary fraction may not exist for a given decimal fraction. For example, the decimal fraction 0.45 used in Figure 9.29 does not have an exact binary equivalent. This is obvious from the pattern developing in the figure. In such cases, the binary fraction is generated to some desired level of accuracy. Of course, some decimal fractions have an exact binary representation. For example, the decimal fraction 0.25 has a binary equivalent of 0.01.

9.9 Concluding Remarks

Computer arithmetic poses several interesting logic design problems. This chapter discussed some of the techniques that have proven useful in designing binary arithmetic units. The carry-lookahead technique is one of the major ideas in high-performance adder design. In the design of fast multipliers, bit-pair recoding of the multiplier, derived from the Booth algorithm, reduces the number of summands that must be added to generate the product. The parallel addition of summands using carry-save reduction trees substantially reduces

https://hemanthrajhemu.github.io

Convert $(927.45)_{10}$

$$\frac{927}{2} = 463 + \frac{1}{2} \rightarrow 1 \text{ LSB}$$

$$\frac{463}{2} = 231 + \frac{1}{2} \rightarrow 1$$

$$\frac{231}{2} = 115 + \frac{1}{2} \rightarrow 1$$

$$\frac{115}{2} = 57 + \frac{1}{2} \rightarrow 1$$

$$\frac{57}{2} = 28 + \frac{1}{2} \rightarrow 1$$

$$\frac{28}{2} = 14 + \frac{0}{2} \rightarrow 0$$

$$\frac{14}{2} = 7 + \frac{0}{2} \rightarrow 0$$

$$\frac{7}{2} = 3 + \frac{1}{2} \rightarrow 1$$

$$\frac{3}{2} = 1 + \frac{1}{2} \rightarrow 1$$

$$\frac{1}{2} = 0 + \frac{1}{2} \rightarrow 1 \text{ MSB}$$

$$0.45 \times 2 = 0.90 \rightarrow 0 \text{ MSB}$$

$$0.90 \times 2 = 1.80 \rightarrow 1$$

$$0.80 \times 2 = 1.60 \rightarrow 1$$

$$0.60 \times 2 = 1.20 \rightarrow 1$$

$$0.20 \times 2 = 0.40 \rightarrow 0$$

$$0.40 \times 2 = 0.80 \rightarrow 0$$

$$0.80 \times 2 = 1.60 \rightarrow 1 \text{ LSB}$$

$$(927.45)_{10} = (1110011111.0111001\ldots)_2$$

Figure 9.29 Conversion from decimal to binary.

https://hemanthrajhemu.github.io

the time needed to add the summands. The important IEEE floating-point number representation standard was described, and rules for performing the four standard operations were given.

9.10 SOLVED PROBLEMS

This section presents some examples of the types of problems that a student may be asked to solve, and shows how such problems can be solved.

Example 9.1 **Problem:** How many logic gates are needed to build the 4-bit carry-lookahead adder shown in Figure 9.4?

Solution: Each B cell requires 3 gates as shown in Figure 9.4*a*. Hence, 12 gates are needed for all four B cells.

The carries c_1, c_2, c_3, and c_4, produced by the carry-lookahead logic, require 2, 3, 4, and 5 gates, respectively, according to the four logic expressions in Section 9.2.1. The carry-lookahead logic also produces G_0^I, using 4 gates, and P_0^I, using 1 gate, as also shown in Section 9.2.1. Hence, a total of 19 gates are needed to implement the carry-lookahead logic.

The complete 4-bit adder requires $12 + 19 = 31$ gates, with a maximum fan-in of 5.

Example 9.2 **Problem:** Assuming 6-bit 2's-complement number representation, multiply the multiplicand A = 110101 by the multiplier B = 011011 using both the normal Booth algorithm and the bit-pair recoding Booth algorithm, following the pattern used in Figure 9.15.

Solution: The multiplications are performed as follows:

(*a*) Normal Booth algorithm

						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	
1	1	1	1	1	1	0	1	0	1		
0	0	0	0	0	1	0	1	1			
							0				
1	1	1	0	1	0	1					
1	1	1	0	1	1	0	1	0	1	1	1

https://hemanthrajhemu.github.io

(*b*) Bit-pair recoding Booth algorithm

						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	0	1					
1	1	1	0	1	1	0	1	0	1	1	1

Example 9.3

Problem: How many levels of 4-2 reducers are needed to reduce k summands to 2 in a reduction tree? How many levels are needed if 3-2 reducers are used?

Solution: Let the number of levels be L.

For 4-2 reducers, we have

$$k(1/2)^L = 2$$

Take logarithms to the base 2 of each side of this equation to derive

$$log_2 k - L = 1$$

or

$$L = log_2 k - 1$$

For 3-2 reducers, we have

$$k(2/3)^L = 2$$

As above, taking logarithms to the base 2, we derive

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

These expressions are only approximations unless the number of input summands to each level is a multiple of 4 in the case of 4-2 reduction, or is a multiple of 3 in the case of 3-2 reduction.

Example 9.4

Problem: Convert the decimal fraction 0.1 to a binary fraction. If the conversion is not exact, give the binary fraction approximation to 8 bits after the binary point using each of the three truncation methods discussed in Section 9.7.2.

Solution: Use the conversion method given in Section 9.8. Multiplying the decimal fraction 0.1 by 2 repeatedly, as shown in Figure 9.29, generates the sequence of bits

https://hemanthrajhemu.github.io

0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, . . . to the left of the decimal point, which continues indefinitely, repeating the pattern 0, 0, 1, 1. Hence, the conversion is not exact.

- Truncation by chopping gives 0.00011001
- Truncation by von Neumann rounding gives 0.00011001
- Truncation by rounding gives 0.00011010

Example 9.5 **Problem:** Consider the following 12-bit floating-point number representation format that is manageable for working through numerical exercises. The first bit is the sign of the number. The next five bits represent an excess-15 exponent for the scale factor, which has an implied base of 2. The last six bits represent the fractional part of the mantissa, which has an implied 1 to the left of the binary point.

Perform Subtract and Multiply operations on the operands

$A =$	0	10001	011011
$B =$	1	01111	101010

which represent the numbers

$$A = 1.011011 \times 2^2$$

and

$$B = -1.101010 \times 2^0$$

Solution: The required operations are performed as follows:

- Subtraction
  According to the Add/Subtract rule in Section 9.7.1, we perform the following four steps:

  1. Shift the mantissa of B to the right by two bit positions, giving 0.01101010.
  2. Set the exponent of the result to 10001.
  3. Subtract the mantissa of B from the mantissa of A by adding mantissas, because B is negative, giving

$$\begin{array}{rcccccccccc} & 1 & . & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ + & 0 & . & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ \hline & 1 & . & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{array}$$

     and set the sign of the result to 0 (positive).

  4. The result is in normalized form, but the fractional part of the mantissa needs to be truncated to six bits. If this is done by rounding, the two bits to be removed represent the tie case, so we round to the nearest even number by adding 1, obtaining a result mantissa of 1.110110. The answer is

$A - B =$	0	10001	110110

https://hemanthrajhemu.github.io

- Multiplication
  According to the Multiplication rule in Section 9.7.1, we perform the following three steps:

  1. Add the exponents and subtract 15 to obtain 10001 as the exponent of the result.
  2. Multiply mantissas to obtain 10.010110101110 as the mantissa of the result. The sign of the result is set to 1 (negative).
  3. Normalize the resulting mantissa by shifting it to the right by one bit position. Then add 1 to the exponent to obtain 10010 as the exponent of the result. Truncate the mantissa fraction to six bits by rounding to obtain the answer

$A \times B =$

0	10010	001011

PROBLEMS

9.1 **[M]** A *half adder* is a combinational logic circuit that has two inputs, x and y, and two outputs, s and c, that are the sum and carry-out, respectively, resulting from the binary addition of x and y.

(*a*) Design a half adder as a two-level AND-OR circuit.

(*b*) Show how to implement a full adder, as shown in Figure 9.2*a*, by using two half adders and external logic gates, as necessary.

(*c*) Compare the longest logic delay path through the network derived in part (*b*) to that of the logic delay of the adder network shown in Figure 9.2*a*.

9.2 **[M]** The 1's-complement and 2's-complement binary representation methods are special cases of the $(b-1)$'s-complement and b's-complement representation techniques in base b number systems. For example, consider the decimal system. The sign-and-magnitude values +526, −526, +70, and −70 have 4-digit signed-number representations in each of the two complement systems, as shown in Figure P9.1. The 9's-complement is formed by

Representation	Examples			
Sign and magnitude	+526	–526	+70	–70
9's complement	0526	9473	0070	9929
10's complement	0526	9474	0070	9930

Figure P9.1 Signed numbers in base 10 used in Problem 9.2.

https://hemanthrajhemu.github.io

taking the complement of each digit position with respect to 9. The 10's-complement is formed by adding 1 to the 9's-complement. In each of the latter two representations, the leftmost digit is zero for a positive number and 9 for a negative number.

Now consider the base-3 (ternary) system, in which the unsigned, 5-digit number $t_4t_3t_2t_1t_0$ has the value $t_4 \times 3^4 + t_3 \times 3^3 + t_2 \times 3^2 + t_1 \times 3^1 + t_0 \times 3^0$, with $0 \le t_i \le 2$. Express the ternary sign-and-magnitude numbers +11011, −10222, +2120, −1212, +10, and −201 as 6-digit, signed, ternary numbers in the 3's-complement system.

9.3 **[M]** Represent each of the decimal values 56, −37, 122, and −123 as signed 6-digit numbers in the 3's-complement ternary format, perform addition and subtraction on them in all possible pairwise combinations, and state whether or not arithmetic overflow occurs for each operation performed. (See Problem 9.2 for a definition of the ternary number system, and use a technique analogous to that given in Section 9.8 for decimal-to-ternary integer conversion.)

9.4 **[M]** A modulo 10 adder is needed for adding BCD digits. Modulo 10 addition of two BCD digits, $A = A_3A_2A_1A_0$ and $B = B_3B_2B_1B_0$, can be achieved as follows: Add A to B (binary addition). Then, if the result is an illegal code that is greater than or equal to 10_{10}, add 6_{10}. (Ignore overflow from this addition.)

(*a*) When is the output carry equal to 1?

(*b*) Show that this algorithm gives correct results for:

(1) $A = 0101$ and $B = 0110$

(2) $A = 0011$ and $B = 0100$

(*c*) Design a BCD digit adder using a 4-bit binary adder and external logic gates as needed. The inputs are $A_3A_2A_1A_0$, $B_3B_2B_1B_0$, and a carry-in. The outputs are the sum digit $S_3S_2S_1S_0$ and the carry-out. A cascade of such blocks can form a ripple-carry BCD adder.

9.5 **[E]** Show that the logic expression $c_n \oplus c_{n-1}$ is a correct indicator of overflow in the addition of 2's-complement integers by using an appropriate truth table.

9.6 **[E]** Use appropriate parts of the solution in Example 9.1 to calculate how many logic gates are needed to build the 16-bit carry-lookahead adder shown in Figure 9.5.

9.7 **[M]** Carry-lookahead adders and their delay are investigated in this problem.

(*a*) Design a 64-bit adder that uses four of the 16-bit carry-lookahead adders shown in Figure 9.5 along with additional logic circuits to generate c_{16}, c_{32}, c_{48}, and c_{64}, from c_0 and the G_i^{II} and P_i^{II} variables shown in the figure. What is the relationship of the additional circuits to the carry-lookahead logic circuits in the figure?

(*b*) Show that the delay through the 64-bit adder is 12 gate delays for s_{63} and 7 gate delays for c_{64}, as claimed at the end of Section 9.2.1.

(*c*) Compare the gate delays to produce s_{31} and c_{32} in the 64-bit adder of part (*a*) to the gate delays for the same variables in the 32-bit adder built from a cascade of two 16-bit adders, as discussed in Section 9.2.1.

9.8 **[M]** Show that the worst case delay through an $n \times n$ array of the type shown in Figure 9.6*b* is $6(n - 1) - 1$ gate delays, as claimed in Section 9.3.1.

https://hemanthrajhemu.github.io

9.9 **[E]** Multiply each of the following pairs of signed 2's-complement numbers using the Booth algorithm. In each case, assume that A is the multiplicand and B is the multiplier.

(a) $A = 010111$ and $B = 110110$

(b) $A = 110011$ and $B = 101100$

(c) $A = 001111$ and $B = 001111$

9.10 **[M]** Repeat Problem 9.9 using bit-pair recoding of the multiplier.

9.11 **[M]** Indicate generally how to modify the circuit diagram in Figure 9.7a to implement multiplication of 2's-complement n-bit numbers using the Booth algorithm, by clearly specifying inputs and outputs for the Control sequencer and any other changes needed around the adder and register A.

9.12 **[M]** Extend the Figure 9.14b table to 16 rows, indicating how to recode three multiplier bits: $i + 2$, $i + 1$, and i. Can all required versions of the multiplicand selected at position i be generated by shifting and/or negating the multiplicand M? If not, what versions cannot be generated this way, and for what cases are they required?

9.13 **[M]** If the product of two n-bit numbers in 2's-complement representation can be represented in n bits, the manual multiplication algorithm shown in Figure 9.6a can be used directly, treating the sign bits the same as the other bits. Try this on each of the following pairs of 4-bit signed numbers:

(a) Multiplicand = 1110 and Multiplier = 1101

(b) Multiplicand = 0010 and Multiplier = 1110

Why does this work correctly?

9.14 **[D]** An integer arithmetic unit that can perform addition and multiplication of 16-bit unsigned numbers is to be used to multiply two 32-bit unsigned numbers. All operands, intermediate results, and final results are held in 16-bit registers labeled R_0 through R_{15}. The hardware multiplier multiplies the contents of R_i (multiplicand) by R_j (multiplier) and stores the double-length 32-bit product in registers R_j and R_{j+1}, with the low-order half in R_j. When $j = i - 1$, the product overwrites both operands. The hardware adder adds the contents of R_i and R_j and puts the result in R_j. The input carry to an Add operation is 0, and the input carry to an Add-with-carry operation is the contents of a carry flag C. The output carry from the adder is always stored in C.

Specify the steps of a procedure for multiplying two 32-bit operands in registers R_1, R_0, and R_3, R_2, high-order halves first, leaving the 64-bit product in registers R_{15}, R_{14}, R_{13}, and R_{12}. Any of the registers R_{11} through R_4 may be used for intermediate values, if necessary. Each step in the procedure can be a multiplication, or an addition, or a register transfer operation.

9.15 **[M]** Delay in multiplier arrays is investigated in this problem.

(a) Calculate the delay, in terms of full-adder block delays, in producing product bit p_7 in each of the 4×4 multiplier arrays in Figure 9.16. Ignore the AND gate delay to generate all $m_i q_j$ products at the beginning.

(b) Develop delay expressions for each of the arrays in Figure 9.16 in terms of n for the $n \times n$ case, as an extension of part (a) of the problem. Then use these expressions to calculate delay for the 32×32 case for each array.

https://hemanthrajhemu.github.io

9.16 **[M]** Tree depth for carry-save reduction is analyzed in this problem.

(*a*) How many 3-2 reduction levels are needed to reduce 16 summands to 2 using a pattern similar to that shown in Figure 9.19?

(*b*) Repeat part (*a*) for reducing 32 summands to 2 to show that the claim of 8 levels in Section 9.5.3 is correct.

(*c*) Compare the exact answers in parts (*a*) and (*b*) to the results obtained by using the approximation developed in Example 9.3 in Section 9.10.

9.17 **[M]** Tree reduction of summands using 3-2 and 4-2 reducers was described in Sections 9.5.3 and 9.5.4. It is also possible to perform 7-3 reductions on each reduction level. When only three summands remain, a 3-2 reduction is performed, followed by addition of the final two summands.

(*a*) How many 7-3 reduction levels are needed to reduce 32 summands to three? Compare this to the seven levels needed to reduce 32 summands to three when using 3-2 reductions.

(*b*) Example 9.3 in Section 9.10 shows that $log_2 k - 1$ levels of 4-2 reduction are needed to reduce k summands to 2 in a reduction tree. How many levels of 7-3 reduction are needed to reduce k summands to 3?

9.18 **[M]** Show how to implement a 4-2 reducer by using two 3-2 reducers. The truth table for this implementation is different from that shown in Figure 9.21.

9.19 **[E]** Using manual methods, perform the operations $A \times B$ and $A \div B$ on the 5-bit unsigned numbers $A = 10101$ and $B = 00101$.

9.20 **[M]** Show how the multiplication and division operations in Problem 9.19 would be performed by the hardware in Figures 9.7*a* and 9.23, respectively, by constructing charts similar to those in Figures 9.7*b* and 9.25.

9.21 **[D]** In Section 9.7, we used the practical-sized 32-bit IEEE standard format for floating-point numbers. Here, we use a shortened format that retains all the pertinent concepts but is manageable for working through numerical exercises. Consider that floating-point numbers are represented in a 12-bit format as shown in Figure P9.2. The scale factor has an implied base of 2 and a 5-bit, excess-15 exponent, with the two end values of 0 and 31 used to signify exact 0 and infinity, respectively. The 6-bit mantissa is normalized as in the IEEE format, with an implied 1 to the left of the binary point.

(*a*) Represent the numbers $+1.7$, -0.012, $+19$, and $\frac{1}{8}$ in this format.

(*b*) What are the smallest and largest numbers representable in this format?

(*c*) How does the range calculated in part (*b*) compare to the ranges of a 12-bit signed integer and a 12-bit signed fraction?

(*d*) Perform Add, Subtract, Multiply, and Divide operations on the operands

$A =$	0	10000	011011
$B =$	1	01110	101010

https://hemanthrajhemu.github.io

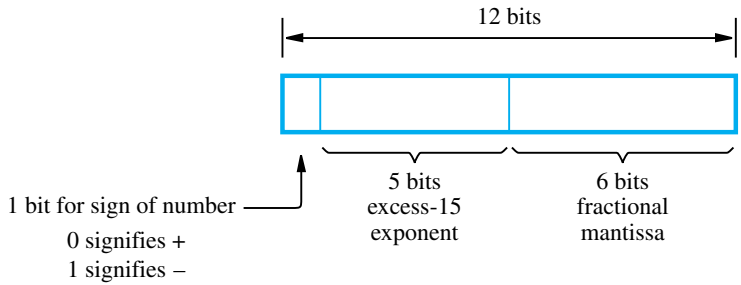


Figure P9.2 Floating-point format used in Problem 9.21.

9.22 **[D]** Consider a 16-bit, floating-point number in a format similar to that discussed in Problem 9.21, with a 6-bit exponent and a 9-bit mantissa fraction. The base of the scale factor is 2 and the exponent is represented in excess-31 format.

(*a*) Add the numbers A and B, formatted as follows:

$A =$	0	100001	111111110
$B =$	0	011111	001010101

Give the answer in normalized form. Remember that an implicit 1 is to the left of the binary point but is not included in the A and B formats. Use rounding as the truncation method when producing the final mantissa.

(*b*) Using decimal numbers w, x, y, and z, express the magnitude of the largest and smallest (nonzero) values representable in the preceding normalized floating-point format. Use the following form:

$$\text{Largest} = w \times 2^{x}$$
$$\text{Smallest} = y \times 2^{-z}$$

9.23 **[M]** How does the excess-x representation for exponents of the scale factor in the floating-point number representation of Figure 9.26*a* facilitate the comparison of the relative sizes of two floating-point numbers? (Hint: Assume that a combinational logic network that compares the relative sizes of two, 32-bit, unsigned integers is available. Use this network, along with external logic gates, as necessary, to design the required network for the comparison of floating-point numbers.)

9.24 **[D]** In Problem 9.21(*a*), conversion of the simple decimal numbers into binary floating-point format is straightforward. However, if the decimal numbers are given in floating-point format, conversion is not straightforward because we cannot separately convert the mantissa and the exponent of the scale factor because $10^{x} = 2^{y}$ does not, in general, allow both x and y to be integers. Suppose a table of binary, floating-point numbers t_i, such that $t_i = 10^{x_i}$ for x_i in the representable range, is stored in a computer. Give a procedure in general terms for

https://hemanthrajhemu.github.io

converting a given decimal floating-point number into binary floating-point format. You may use both the integer and floating-point instructions available in the computer.

9.25 **[D]** Construct an example to show that three guard bits are needed to produce the correct answer when two positive numbers are subtracted.

9.26 **[M]** Derive logic expressions that specify the Add/Sub and S_R outputs of the combinational CONTROL network in Figure 9.28.

9.27 **[M]** If gate fan-in is limited to four, how can the SHIFTER in Figure 9.28 be implemented combinationally?

9.28 **[M]** Sketch a logic-gate network that implements the multiplexer MUX in Figure 9.28.

9.29 **[M]** Relate the structure of the SWAP network in Figure 9.28 to your solution to Problem 9.28.

9.30 **[M]** How can the leading zeros detector in Figure 9.28 be implemented combinationally?

9.31 **[M]** The mantissa adder-subtractor in Figure 9.28 operates on positive, unsigned binary fractions and must produce a sign-and-magnitude result. In the discussion accompanying Figure 9.28, we state that 1's-complement arithmetic is convenient because of the required format for input and output operands. When adding two signed numbers in 1's-complement notation, the carry-out from the sign position must be added to the result to obtain the correct signed answer. This is called *end-around carry correction*. Consider the two examples in Figure P9.3, which illustrate addition using signed, 4-bit encodings of operands and answers in the 1's-complement system.

The 1's-complement arithmetic system is convenient when a sign-and-magnitude result is to be generated because a negative number in 1's-complement notation can be converted to sign-and-magnitude form by complementing the bits to the right of the sign-bit position. Using 2's-complement arithmetic, addition of +1 is needed to convert a negative value into sign-and-magnitude notation. If a carry-lookahead adder is used, it is possible to incorporate the end-around carry operation required by 1's-complement arithmetic into the lookahead logic. With this discussion as a guide, give the complete design of the 1's-complement adder/subtractor required in Figure 9.28.

9.32 **[M]** Signed binary fractions in 2's-complement representation are discussed in Section 1.4.2.

(*a*) Express the decimal values 0.5, −0.123, −0.75, and −0.1 as signed 6-bit fractions. (See Section 9.8 for decimal-to-binary fraction conversion.)

(3)			0		0		1		1							
+(−5)	+	$\boxed{0}$	1	0	0	1	1	0	0	0						
−2			1		1		0		1							
									0							
			1		1		0		1							

(6)			0		1		1		0							
+(−3)	+	$\boxed{1}$	1	0	1	1	0	0	0	0						
3			0		0		1		0							
									1							
			0		0		1		1							

Figure P9.3 1's-complement addition used in Problem 9.31.

https://hemanthrajhemu.github.io

(*b*) What is the maximum representation error, *e*, involved in using only 5 significant bits after the binary point?

(*c*) Calculate the number of bits needed after the binary point so that the representation error *e* is less than 0.1, 0.01, or 0.001, respectively.

9.33 **[E]** Which of the four 6-bit answers to Problem 9.32(*a*) are not exact? For each of these cases, give the three 6-bit values that correspond to the three types of truncation defined in Section 9.7.2.

REFERENCES

1. A. D. Booth, "A Signed Binary Multiplication Technique," *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 2, part 2, 1951, pp. 236-240.
2. C. S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, February 1964, pp. 14-17.
3. M. R. Santoro and M. A. Horowitz, "SPIM: A Pipelined 64 × 64-bit Iterative Multiplier," *IEEE Journal of Solid-State Circuits*, vol. 24, No.2, April 1989, pp. 487-493.
4. Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-2008, August 2008.

https://hemanthrajhemu.github.io

This page intentionally left blank

https://hemanthrajhemu.github.io