COMPUTER ARITHMETIC

Background Information

- Binary Numbers
  - 2's Complement representation
  - Addition
  - Subtraction

- Arithmetic Logic Unit
  - Contains Adders to perform addition and subtraction

Integer Multiplication

- "Paper and pencil" example

```
Multiplicand          1000
Multiplier          x 1001
                      1000
                     0000
                    0000
                 + 1000
Product            01001000
```

Shift after each step

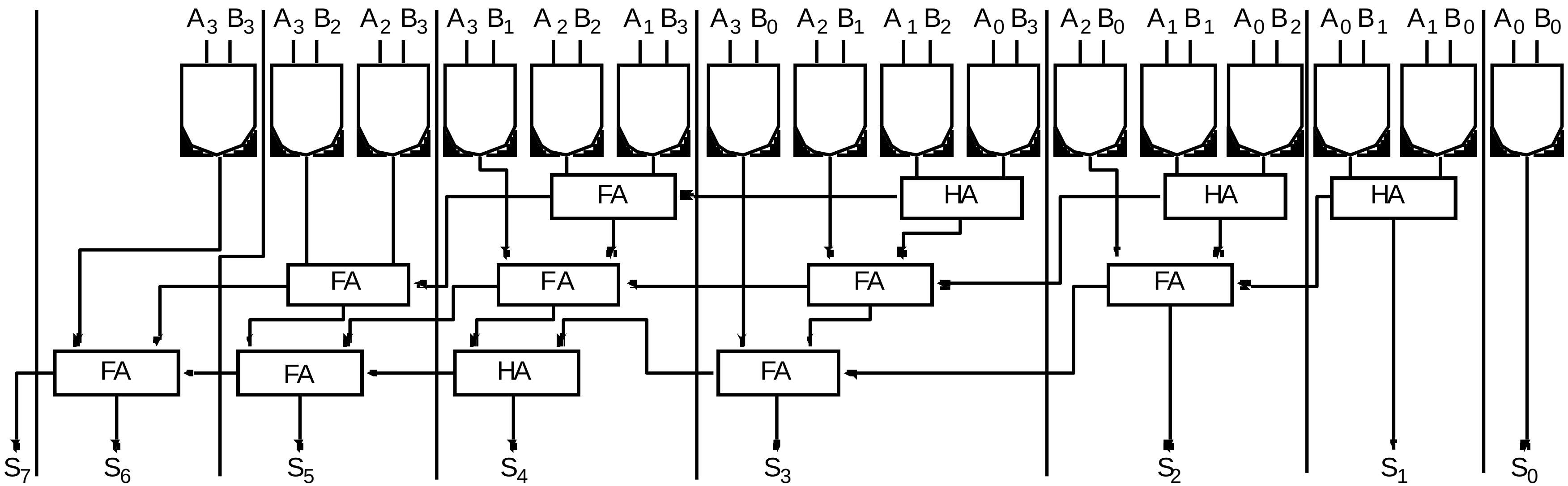
Combinational Multiplier

- Partial product accumulation

				A3	A2	A1	A0
				B3	B2	B1	B0
				A3 B0	A2 B0	A1 B0	A0 B0
			A3 B1	A2 B1	A1 B1	A0 B1	
		A3 B2	A2 B2	A1 B2	A0 B2		
	A3 B3	A2 B3	A1 B3	A0 B3			
S7	S6	S5	S4	S3	S2	S1	S0

Combinational Multiplier

- Partial product accumulation



Note use of parallel carry-outs to form higher order sums

12 Adders, if full adders, this is 6 gates each = 72 gates

16 gates form the partial products

total = 88 gates

Integer Multiplication

- “Paper and pencil” example

```
Multiplicand        1000
Multiplier        x 1001
                    1000
                   0000
                  0000          Shift after each step
               + 1000
Product          01001000
```

Observations

- Number of bits in the product is larger than the number in either the multiplicand or the multiplier.
  - m bits x n bits = m+n bit product
  - Overflow is a possible issue

- Binary rules – “choices”

0 => place 0	(0 x multiplicand)
1 => place a copy	(1 x multiplicand)

- 3 versions of unsigned multiplication hardware
  - successive refinement

Multiplication

- Insight from paper and pencil algorithm
  - Shift the multiplicand left one digit each step
  - With 32 steps in a 32-bit number, we move 32 bits to the left
  - Requires a 64-bit register
  - Place 32 zeroes in the left half (unoccupied half)
    - Unsigned numbers do not require sign extension

  - Multiplicand will be added to the sum in the product register
    - Product register will also be 64 bits
    - Requires a 64 bit ALU to add

Multiplication Hardware Version 1

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg

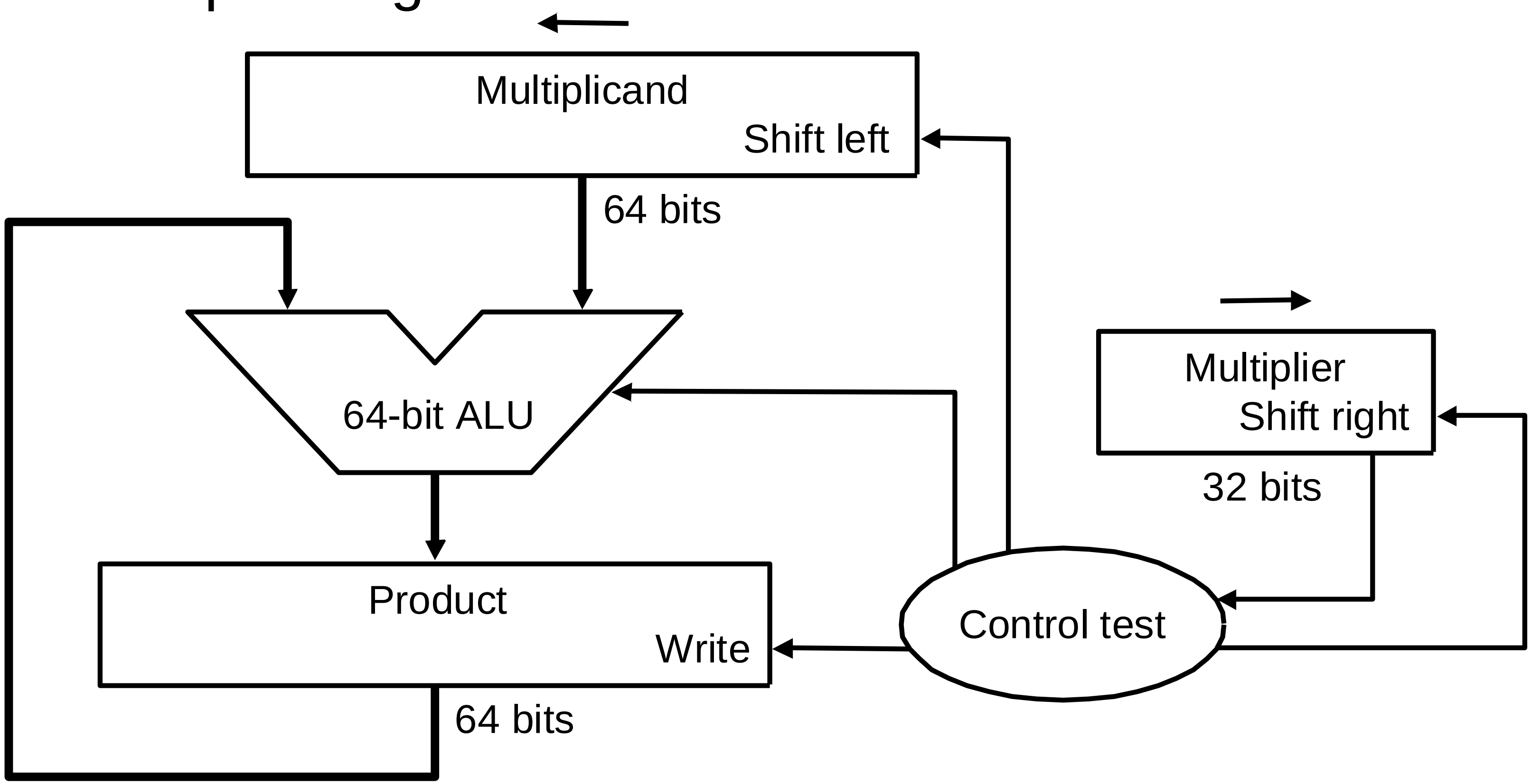


Figure 3.3 from text

Multiplication Algorithm Version 1

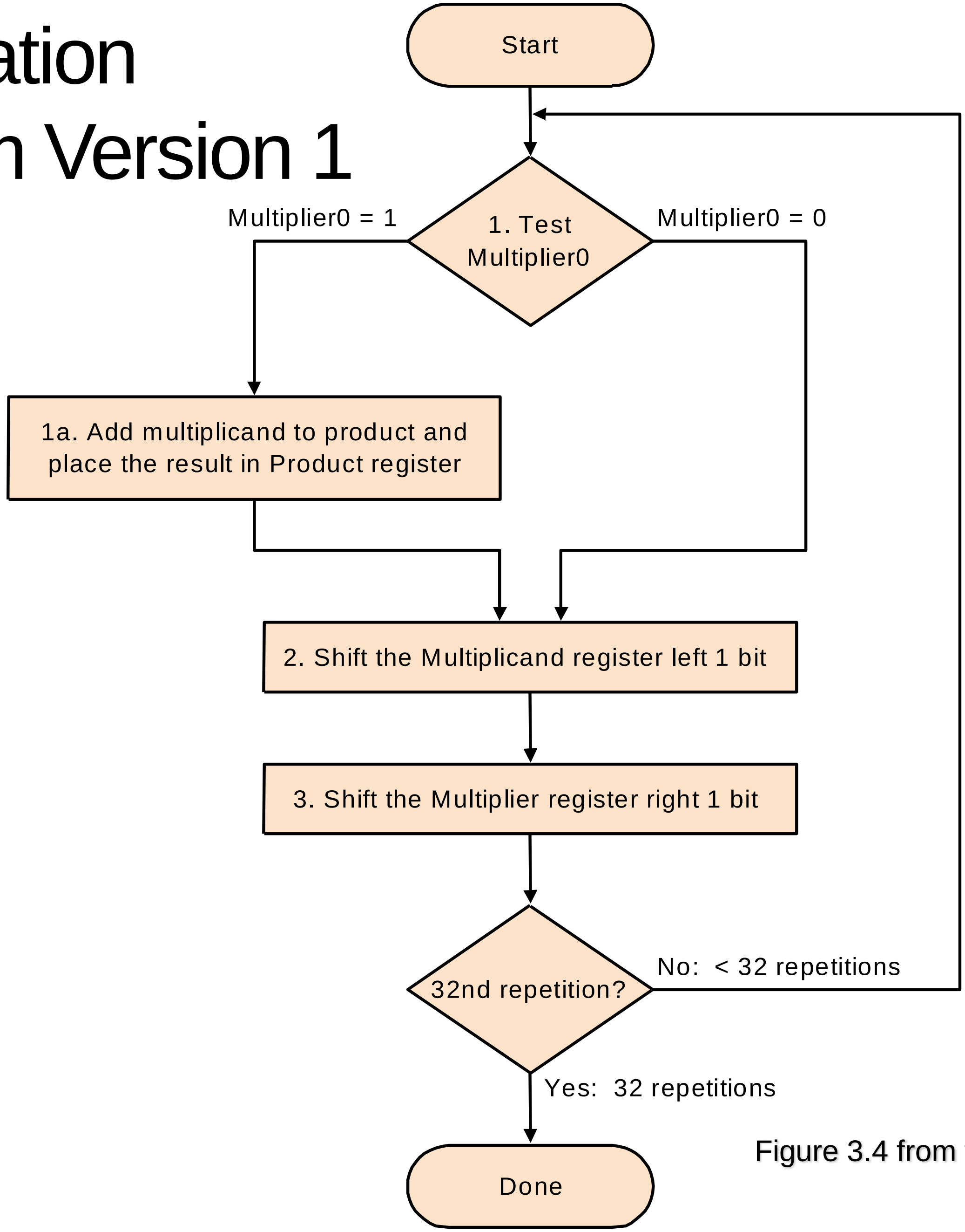


Figure 3.4 from text

Multiplication Example (11x9)

Iter.	Step	Product	Multiplicand	Multiplier	Action
0	0	00000000	00001011	1001	Initialize
1	1a.	00001011	00001011	1001	Add
1	2,3	00001011	00010110	0100	Shifts
2	1	00001011	00010110	0100	Test-no add
2	2,3	00001011	00101100	0010	Shifts
3	1	00001011	00101100	0010	Test-no add
3	2,3	00001011	01011000	0001	Shifts
4	1a.	01100011	01011000	0001	Add
4	2,3	01100011	10110000	0000	Shifts

Multiplication is Time Consuming

- 3 steps per iteration
- 32 iterations
- 96 steps total

Observations on Multiplication Version 1

- Half the bits of the multiplicand are always 0
  - 64-bit adder is wasted
- 0’s inserted in right of multiplicand as shifted
  - LSBs of product never changed once formed
- Instead of shifting the multiplicand to the left we can shift the product to the right
  - Perform some steps in parallel

Multiplication Hardware Version 2

- 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, 32-bit Multiplier reg

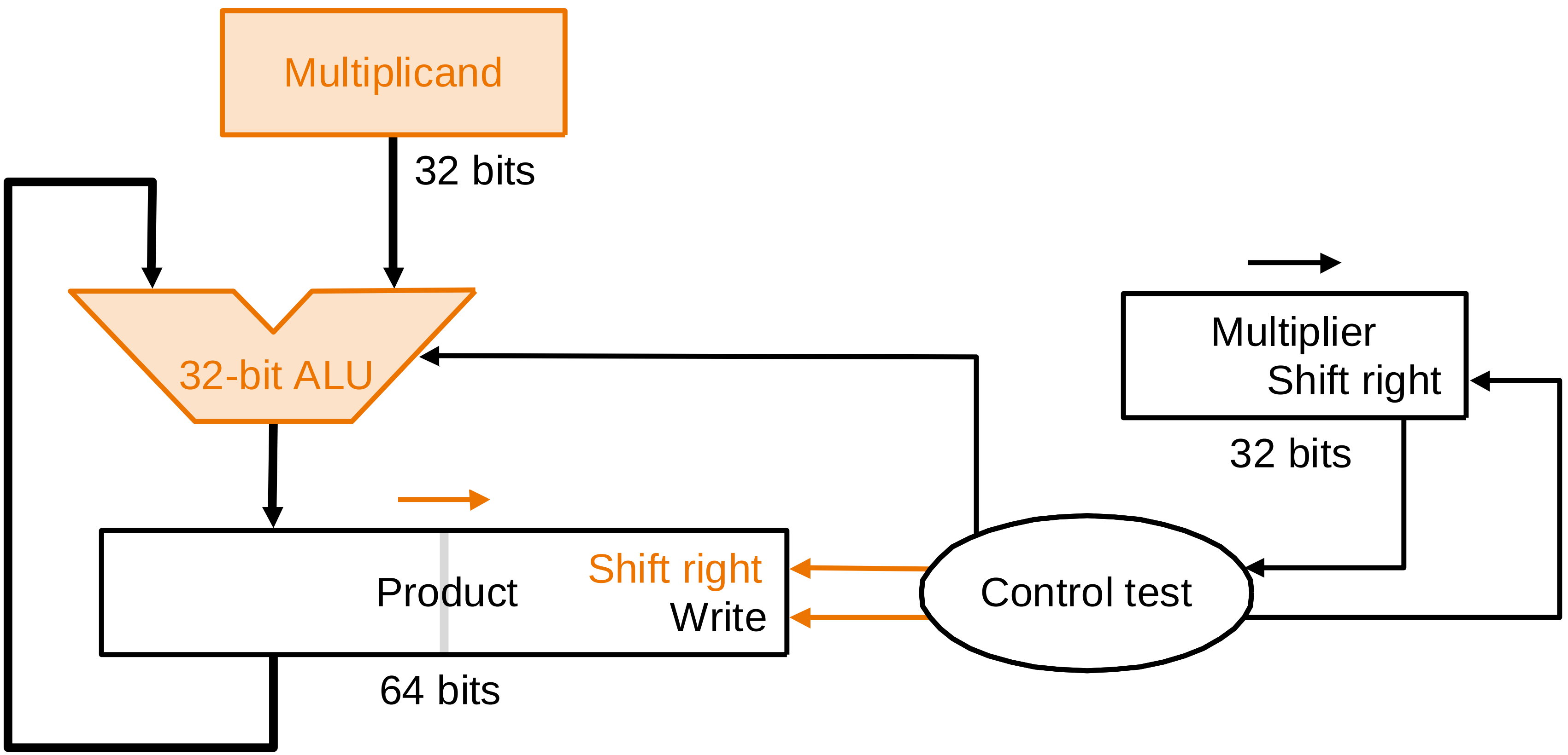


Figure from a previous version of the text

Multiplication Algorithm Version 2

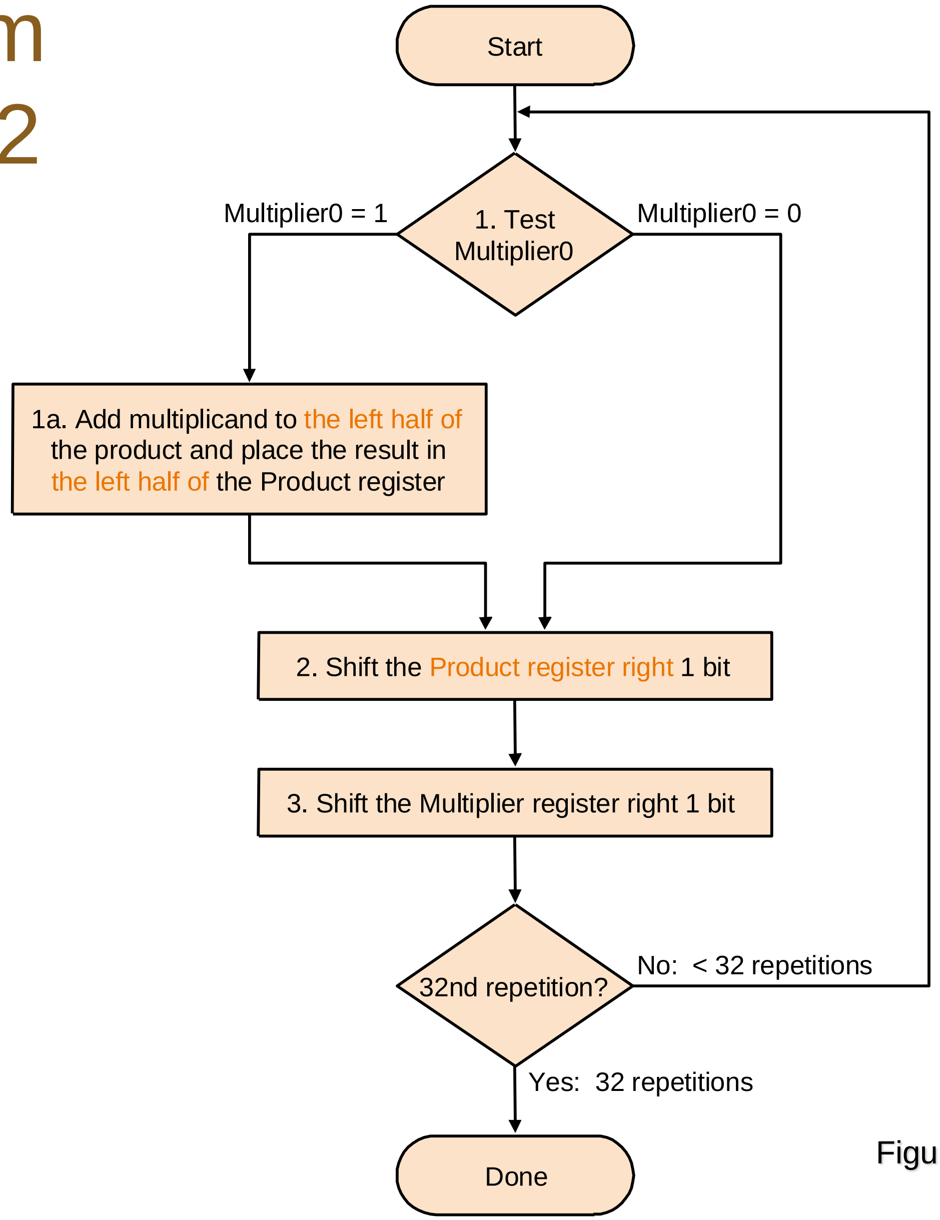


Figure from a previous version of the text

Multiplication Example (11x9)

Iter.	Step	Product	Multiplicand	Multiplier	Action
0	0	00000000	1011	1001	Initialize

Multiplication Example (11x9)

Iter.	Step	Product	Multiplicand	Multiplier	Action
0	0	00000000	1011	1001	Initialize
				Test the LSB of multiplier	
				1 indicates Add	

Multiplication Example (11x9)

Iter.	Step	Product	Multiplicand	Multiplier	Action
0	0	00000000	1011	1001	Initialize
				1001	Add

Add the left half of the product to the multiplicand. Store in left half of product.

Multiplication Example (11x9)

Iter.	Step	Product	Multiplicand	Multiplier	Action
0	0	00000000	1011	1001	Initialize
1	1a.	10110000	1011	1001	Add

Add the left half of the product to the multiplicand. Store in left half of product.

Multiplication Example (11x9)

Iter.	Step	Product	Multiplicand	Multiplier	Action
0	0	00000000	1011	1001	Initialize
1	1a.	10110000	1011	1001	Add

Shift both the product and the multiplier to the right.

Multiplication Example (11x9)

Iter.	Step	Product	Multiplicand	Multiplier	Action
0	0	00000000	1011	1001	Initialize
1	1a.	10110000	1011	1001	Add
1	2, 3	01011000	1011	0100	Shifts

Shift both the product and the multiplier to the right.

Multiplication Example (11x9)

Iter.	Step	Product	Multiplicand	Multiplier	Action
0	0	00000000	1011	1001	Initialize
1	1a.	10110000	1011	1001	Add
1	2,3	01011000	1011	0100	Shifts
2	1	01011000	1011	0100	Test-no add
2	2,3	00101100	1011	0010	Shifts
3	1	00101100	1011	0010	Test-no add
3	2,3	00010110	1011	0001	Shifts
4	1a.	11000110	1011	0001	Add
4	2,3	01100011	1011	0000	Shifts

Multiplication Algorithm Version 2

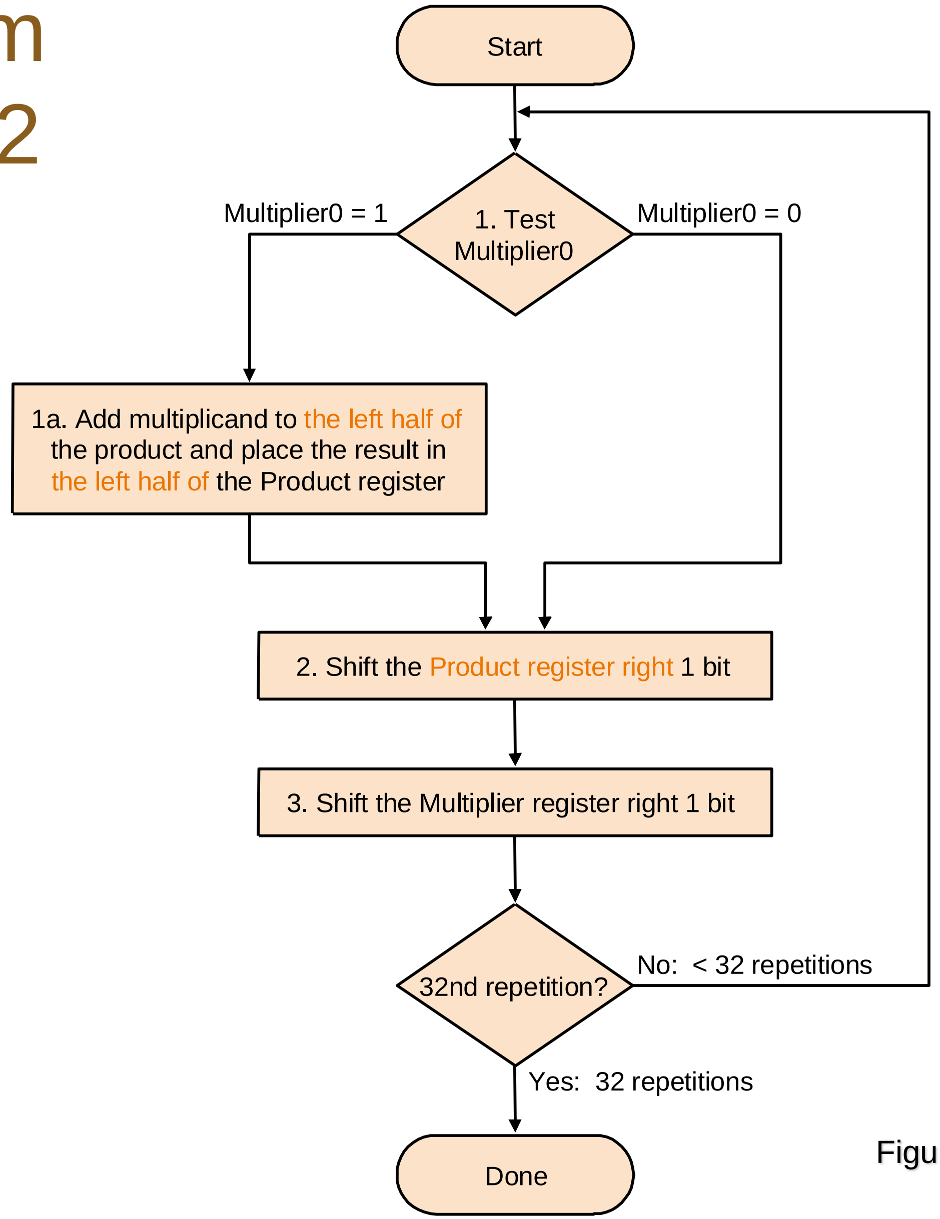


Figure from a previous version of the text

Observation

Product register wastes space (lower half = 0)

Exactly equal to the size of multiplier left

We can combine Multiplier register and Product register

Multiplication Hardware Version 3

- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, (no Multiplier reg)

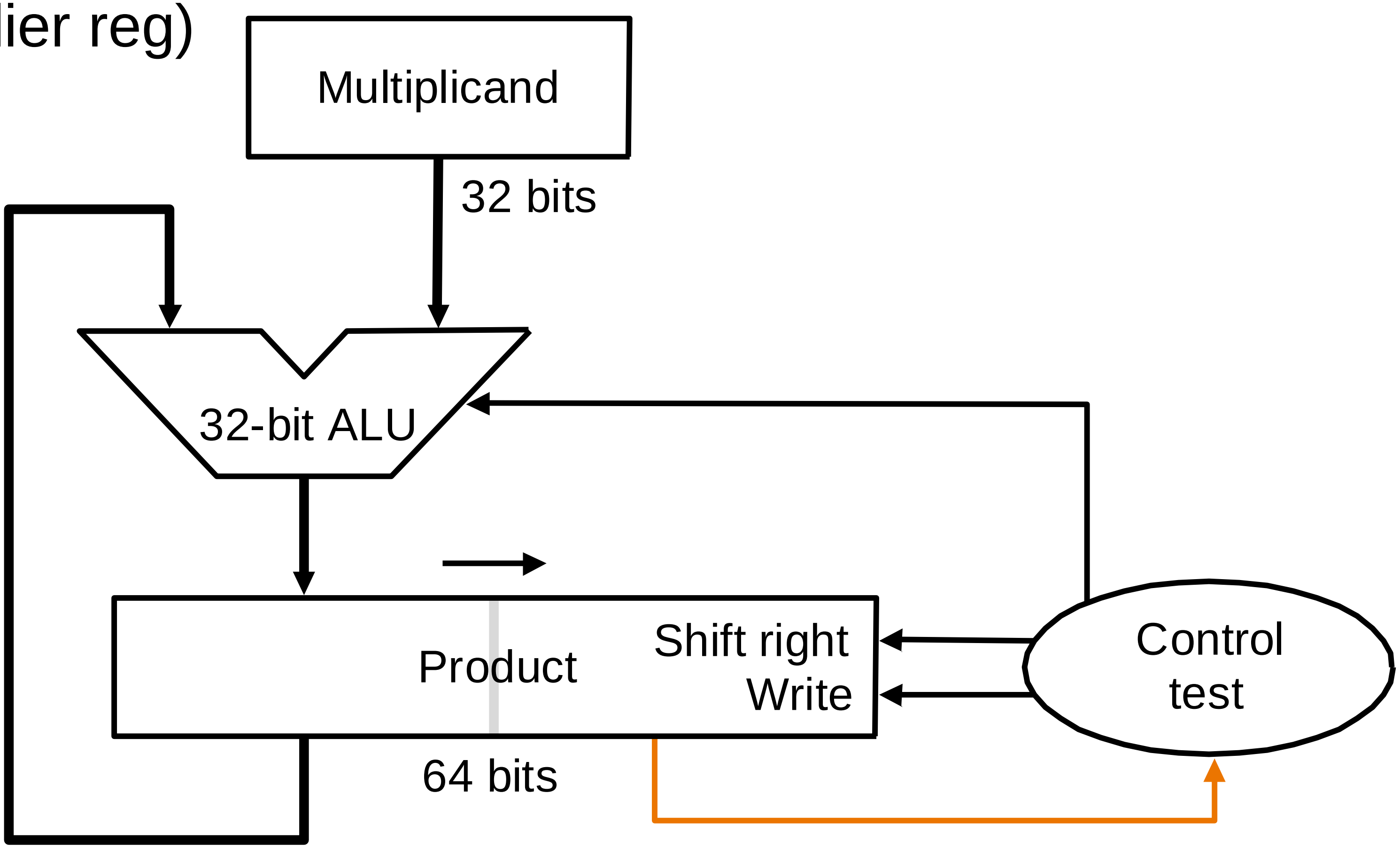


Figure 3.5 from text

Multiplication Algorithm Version 3

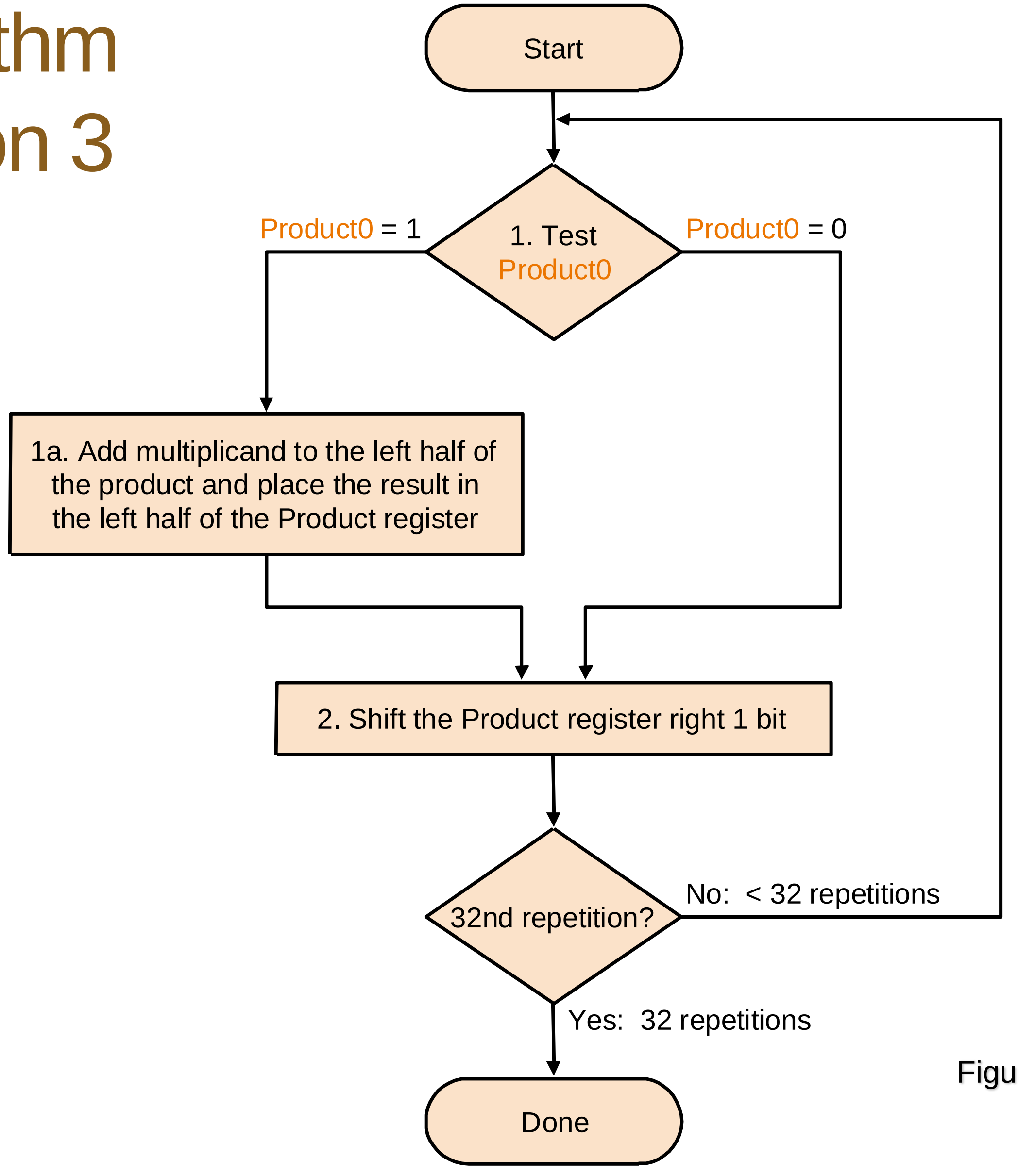


Figure from a previous version of the text

Multiplication Example (11x9)

Iter.	Step	Product	Multiplicand	Action
0	0	0000 $\underline{1001}$	1011	Initialize
1	1a.	1011 $\underline{1001}$	1011	Add
1	2	0101 1$\underline{100}$	1011	Shift
2	1	0101 1$\underline{100}$	1011	Test-no add
2	2	0010 11$\underline{10}$	1011	Shift
3	1	0010 11$\underline{10}$	1011	Test-no add
3	2	0001 011$\underline{1}$	1011	Shift
4	1a.	1100 011$\underline{1}$	1011	Add
4	2	0110 0011	1011	Shift

Note: Multiplier in Product Register is underlined

Multiplying by a Constant

- Some compilers replace multiplies by short constants with a series of shifts and adds. Because one bit to the left represents a number twice as large in base 2, shifting the bits left has the same effect as multiplying by a power of 2.
- Almost every compiler will perform the strength reduction optimization of substituting a left shift for a multiply by a power of 2.

Multiplying by a Constant

- Some compilers replace multiplies by short constants with a series of shifts and adds. Because one bit to the left represents a number twice as large in base 2, shifting the bits left has the same effect as multiplying by a power of 2.
- Almost every compiler will perform the strength reduction optimization of substituting a left shift for a multiply by a power of 2.

- 4 * 2 = 8
- 0100 * 0010 = 1000
- 0100 << 1 = 1000

Multiplying by a Constant

- Some compilers replace multiplies by short constants with a series of shifts and adds. Because one bit to the left represents a number twice as large in base 2, shifting the bits left has the same effect as multiplying by a power of 2.
- Almost every compiler will perform the strength reduction optimization of substituting a left shift for a multiply by a power of 2.

- 2 * 4 = 8
- 0010 * 0100 = 1000
- 0010 << 2 = 1000

Signed Multiplication

- So far, we have multiplied unsigned numbers
- What about signed multiplication?
  - one solution: make both positive
  - leave out the sign bit, run for 31 steps
  - set sign bit negative if signs of inputs differ

Booth’s Algorithm

- multiply two’s complement signed numbers
- uses same hardware as before
- can also be used to reduce the number of steps

Insight for Booth's Algorithm

- Example: 2 x 6 = 0010 x 0110:

```
        0010
x       0110
+       0000   shift (0 in multiplier)
+      0010    add (1 in multiplier)
+     0010     add (1 in multiplier)
+    0000      shift (0 in multiplier)
  00001100
```

- ALU can get same result in more than one way:
  - 6 x = 4x + 2x or 6x = – 2x + 8 x
  - 111 = 1000 – 0001
  - 1111 = 10000 – 00001
  - 1111XXX = 10000XXX – 00001XXX

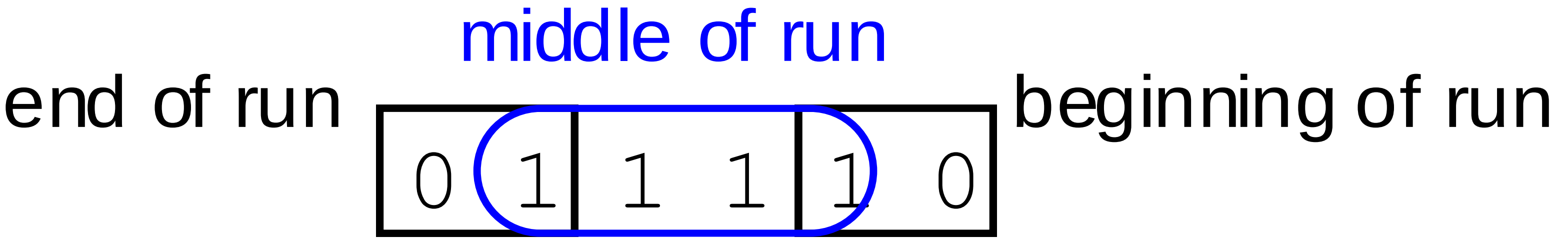
Insight for Booth's Algorithm

- Replace string of 1s in multiplier with
  - initially subtract when we see first 1 (from right)
  - later, add when we see 0 at left end of the string of 1s.
- Example

```
         0010
x        0110
+        0000   shift (0 in multiplier)
-       0010    subtract (first 1 in string)
+      0000     shift (within string of 1s)
+     0010      add (end of string)
     00001100
```

- Effectively: 2 x 6 = 2 x 8 – 2 x 2

Booth's Algorithm



Current	Right	Explanation	Example
1	0	Beginning of a run of 1s	0001111000
1	1	Middle of a run of 1s	0001111000
0	1	End of a run of 1s	0001111000
0	0	Middle of a run of 0s	0001111000

Booth's Algorithm

1. Depending on the current and previous bits, do one of the following:
   00: Middle of a string of 0s, so no arithmetic operations.
   01: End of a string of 1s, so add the multiplicand to the left half of the product.
   10: Beginning of a string of 1s, so subtract the multiplicand from the left half of the product.
   11: Middle of a string of 1s, so no arithmetic operation.
2. As in the previous algorithm, shift the Product register right (arithmetic shift) 1 bit.

Booth’s Example (-5 x -6)

- Multiplicand = –6 = 1010; –Multiplicand = 6 = 0110
- Multiplier = –5 = 1011

Iter.	Step	Product	Last	Action
0	0	0000 101(1 0)	0	Initialize
1	1.10	0110 101(1 0)	0	Start string: Subtract => Add 0110
1	2	0011 010(1 1)	1	Shift arithmetic
2	1.11	0011 010(1 1)	1	Middle string: nothing
2	2	0001 101(0 1)	1	Shift arithmetic
3	1.01	1011 101(0 1)	1	End string: add 1010
3	2	1101 110(1 0)	0	Shift *arithmetic*
4	1.10	0011 110(1 0)	0	Start string: Subtract => add 0110
4	2	0001 1110	1	Shift arithmetic

Notes: 1. Multiplier in Product Register is underlined.
2. Current/previous bits are in parentheses.
3. Previous bit is initialized to 0

Booth's Algorithm

- Originally for speed: Shifts are faster than add
- Key advantage today: Works properly for 2's complement numbers without requiring special fix for sign!

Division: Paper and Pencil

- “Paper and pencil” example

- 20 ÷ 6 = 3 Remainder 2

```
                 00011        Quotient
              ________
Divisor   110 |10100          Dividend
               10
               101
               1010
             -  110
             -------
                 1000
               -  110
               ------
                   10         Remainder
```

Dividend = Quotient * Divisor + Remainder

Division: Paper and Pencil

- "Paper and pencil" example

- 20 ÷ 6 = 3 Remainder 2

```
                00011      Quotient
Divisor    110 |10100      Dividend
                10
                101
                1010
              -  110
                 1000
               -  110
                   10      Remainder
```

Algorithm:

If Partial Remainder > Divisor

then Quotient bit = 1;

Remainder = Remainder – Divisor

else Quotient bit = 0

Shift down next dividend bit

Division Hardware

- Same as Multiplication Hardware!
- 32-bit Divisor reg, 32 -bit ALU, 64-bit Remainder reg
- Dividend stored in remainder register, Quotient formed in remainder register

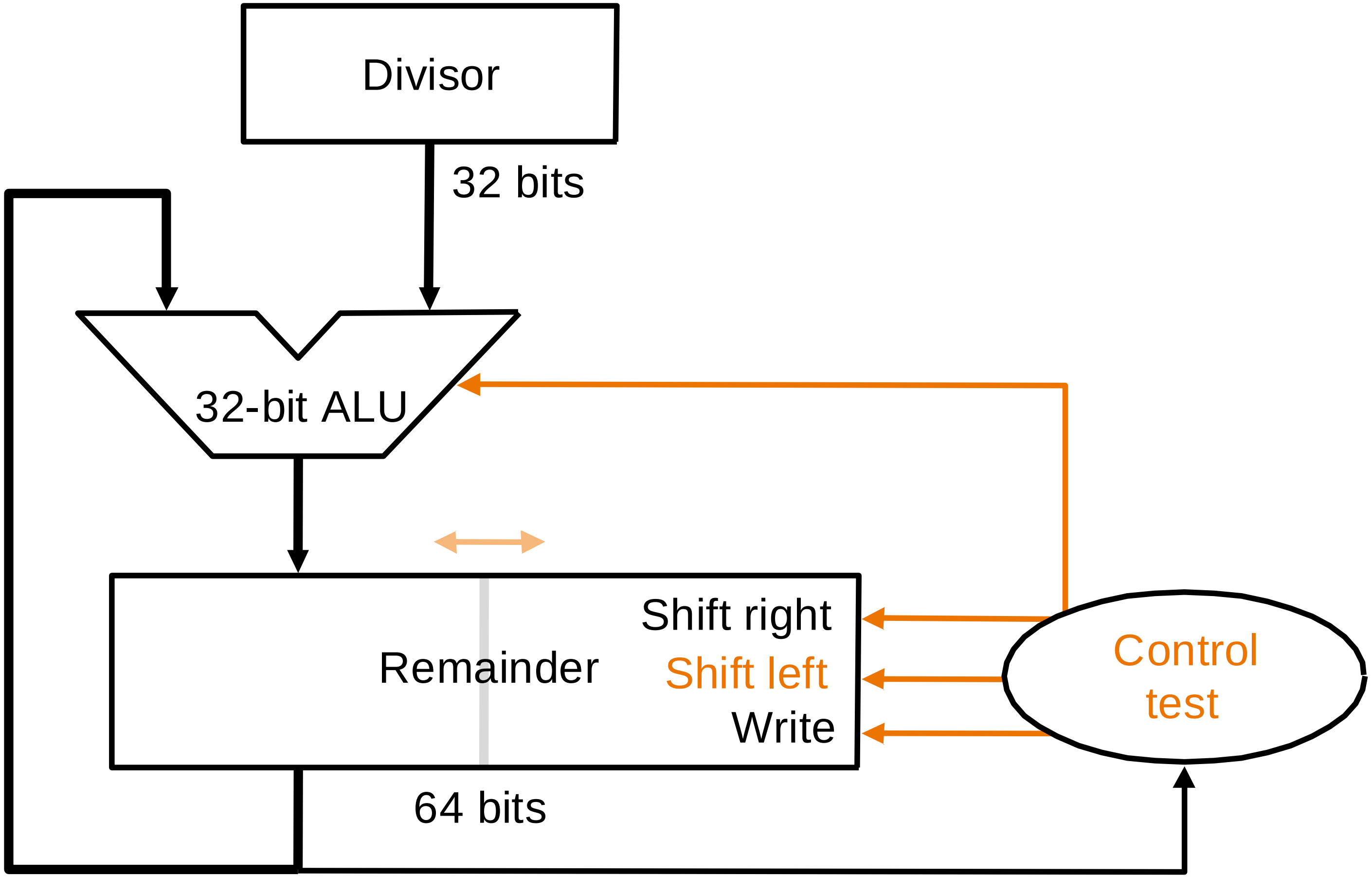
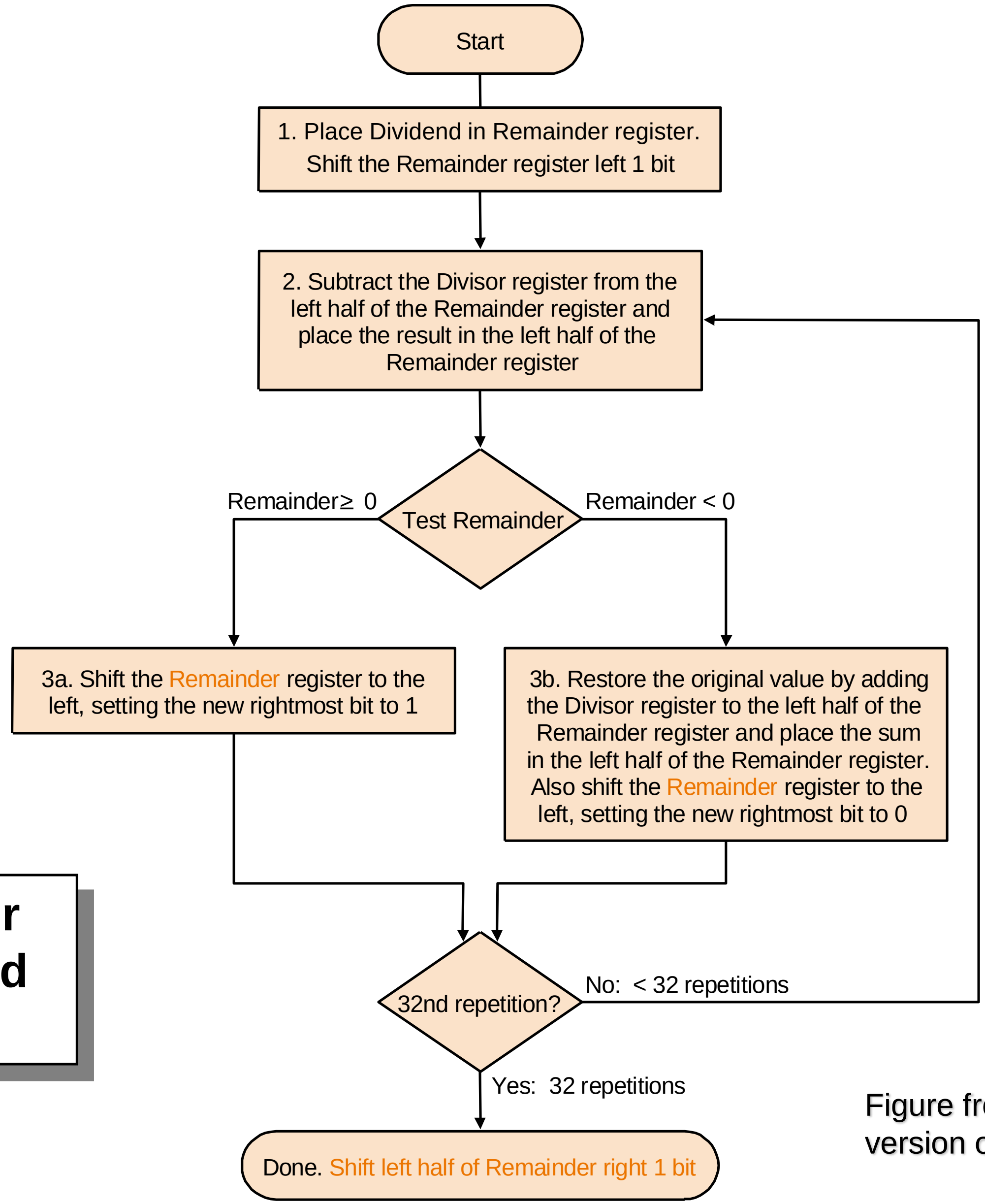


Figure 3.11 from text

Division Algorithm



Takes n Steps for n-bit Quotient and Remainder

Figure from a previous version of the text

Division Example

- Example: 14 ÷ 3 = 4; remainder 2.

Iter	Step	Remainder	Divisor	Action
0	0	0001 1100	0011	Initialize
1	1	1110 1100	0011	Subtract: Remainder<0
1	2b.	0011 1000	0011	Restore; shift in 0
2	1	0000 1000	0011	Subtract; Remainder=0
2	2a.	0001 0001	0011	Shift in a 1
3	1	1110 0001	0011	Subtract: Remainder<0
3	2b.	0010 0010	0011	Restore; shift in 0
4	1	1111 0010	0011	Subtract: Remainder<0
4	2b.	0100 0100	0011	Restore; shift in 0
	3	0010 0100	0011	Shift remainder right
		Rem. Quot.		

Observations on Division Hardware

- Same Hardware as Multiply: just need ALU to add or subtract, and 64-bit register to shift left or shift right
- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide

Signed Division

- Store the signs of the divisor and dividend
- Convert divisor and dividend to positive
- Complement quotient and remainder if necessary
  - Dividend and Remainder are defined to have same sign
  - Quotient negated if Divisor sign and Dividend sign disagree

Beyond Integers

- Real numbers
  - Called “float” values

- Computer arithmetic that supports real numbers is called floating point arithmetic

Exponential Notation

- The following are equivalent representations of 1,234

123,400.0	$\times 10^{-2}$
12,340.0	$\times 10^{-1}$
1,234.0	$\times 10^{0}$
123.4	$\times 10^{1}$
12.34	$\times 10^{2}$
$\boxed{1.234 \times 10^{3}}$	
0.1234	$\times 10^{4}$

The representations differ in that the decimal place – the “point” -- “floats” to the left or right (with the appropriate adjustment in the exponent).

Standards

- Floats are implemented using the IEEE 754 standard
  - found in virtually every computer invented since 1980
  - has greatly improved both the ease of porting floating-point programs and the quality of computer arithmetic.

- IEEE 754 was created to:
  - Simplify exchange of data that includes floating-point numbers
  - Simplify the floating-point arithmetic algorithms
  - Increases the accuracy of the numbers that can be stored
    - Increased accuracy due to normalized scientific notation

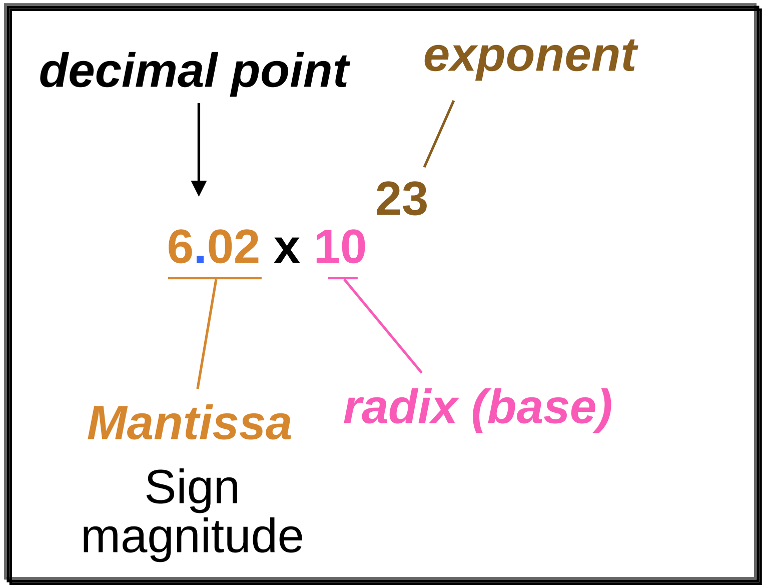
Normalized Scientific Notation

- A number in scientific notation that has no leading 0s is called a normalized number.
  - $1.0_{ten} * 10^{-9}$ is in normalized scientific notation

  - $0.1_{ten} * 10^{-8}$ is not normalized
  - $10.0_{ten} * 10^{-10}$ is not in scientific notation

Floating Point: Scientific Notation

- Number represented as
  - Mantissa
  - Radix (base)
  - Exponent



In a binary number, the radix (or base) is 2 instead of 10. The general form could be written as 1.xxxxxx * 2^{yyyyy}.

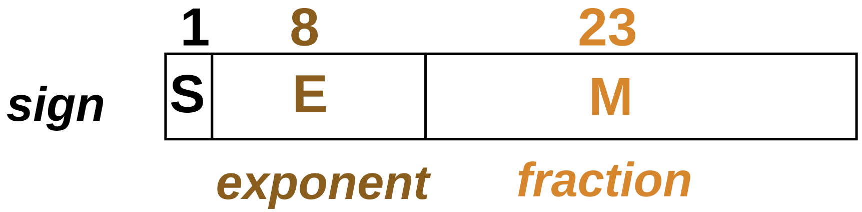
Floating Point: Normalized Scientific Notation

- The mantissa must be normalized: 1.xxxxxx * 2^{yyyyy}
- Always has a 1 in front of the binary point
- This 1 does not need to be stored

- Floating point numbers have an implied "1" on left of the decimal place
  - Mantissa $\rightarrow$ 10100000000000000000000
  - Represents $\rightarrow$ $1.101_2 = 1.625_{10}$

IEEE 754 Standard

- Single precision: 32 bits, consisting of...
  - Sign bit (1 bit)
  - Exponent (8 bits)
  - Mantissa (23 bits)



Normalized binary significand with *hidden* bit (1): 1.M

IEEE 754 Standard

- Single precision: 32 bits, consisting of...
  - Sign bit (1 bit)
  - Exponent (8 bits)
  - Mantissa (23 bits)

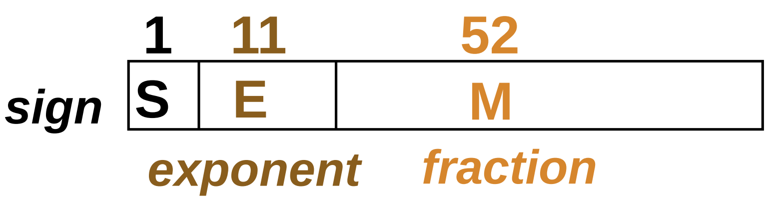
- Fractions almost as small as $2.0_{ten} * 10^{-38}$
- Numbers almost as large as $2.0_{ten} * 10^{38}$

- Overflow may still occur
  - Exponent is too large to be represented
- Underflow may occur
  - Exponent is too small to be represented

IEEE 754 Standard

- Single precision: 32 bits, consisting of...
  - Sign bit (1 bit)
  - Exponent (8 bits)
  - Mantissa (23 bits)

- Double precision: 64 bits, consisting of…
  - Sign bit (1 bit)
  - Exponent (11 bits)
  - Mantissa (52 bits)



Normalized binary significand with *hidden* bit (1): 1.M

Normalization

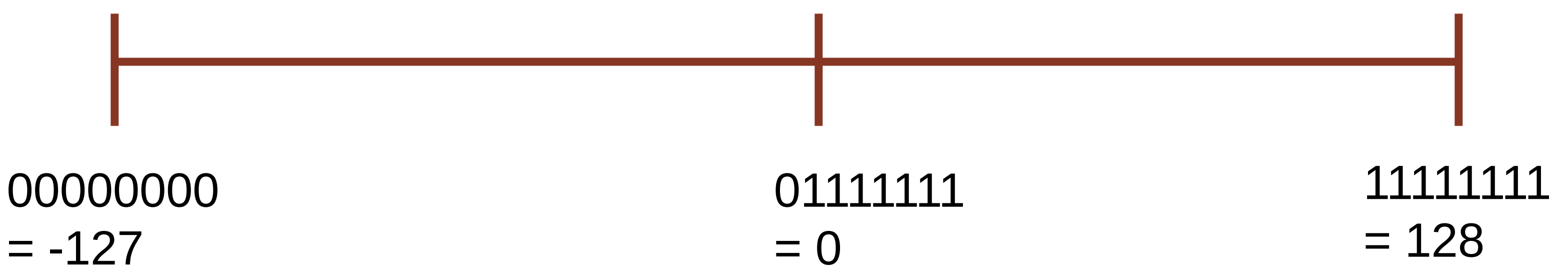
- General form for floating-point numbers: $(-1)^S * (1+M) * 2^E$
- How do we represent zero?
  - E = 0
  - M = 0

Excess Notation

- To include positive (+ve) and negative (–ve) exponents, “excess” notation is used
- Also called biased notation

- Represents the most negative exponent as $0...0_{two}$ and the most positive exponent as $1…1_{two}$.

Single Precision (8-bit Exponent): 00000000 – 11111111 (0-255)



Excess Notation

- The value of the exponent stored is larger than the actual exponent

- Single precision: excess 127
- Double precision: excess 1023

- Each real number is
  $(-1)^S * (1 + \text{Fraction}) * 2^{(\text{Exponent} - \text{Bias})}$

- E.g., excess 127,
  - Exponent $\rightarrow$ `10000111`
  - Represents… `135 - 127 = 8`

Example

- Single precision

0 10000010 11000000000000000000000

1.11_2

130 – 127 = 3

0 = positive mantissa

$$+1.11_2 \times 2^3 = 1110.0_2 = 14.0_{10}$$

Converting from Floating Point

- What decimal value is represented by the following 32-bit floating point number?

$$1100\ 0001\ 0111\ 1011\ 0000\ 0000\ 0000\ 0000_2$$

Converting from Floating Point

- Step 1: find S, E, and M

1	10000010	$11110110000000000000000_2$
S	E	M

1 = negative
0 = positive

Converting from Floating Point

- Step 2: Find “real” exponent, *n*
  - $n = E - 127$

    $= 10000010_2 - 127$

    $= 130 - 127$

    $= 3$

Converting from Floating Point

- Step 3: Put S, M, and *n* together to form binary result
  - Don't forget the implied "1." on the left of the mantissa.

$$-1.1111011_2 \text{ x } 2^n =$$

$$-1.1111011_2 \text{ x } 2^3 =$$

$$-1111.1011_2$$

Converting from Floating Point

- Step 4: Express result in decimal

$$-\underbrace{1111}_{-15}.1011_2$$

$$\begin{aligned} 2^{-1} &= 0.5 \\ 2^{-3} &= 0.125 \\ 2^{-4} &= \underline{0.0625} \\ & \ 0.6875 \end{aligned}$$

Answer: -15.6875

Converting to Floating Point

- Express 36.5625_{10} as a 32-bit floating point number

Converting to Floating Point

- Step 1: Express original value in binary

$$36.5625_{10} = 100100.1001_2$$

36 = 2 * 18 + 0	.5625 * 2 = 1.125
18 = 2 * 9 + 0	.125 * 2 = 0.25
9 = 2 * 4 + 1	.25 * 2 = 0.5
4 = 2 * 2 + 0	.5 * 2 = 1.0
2 = 2 * 1 + 0	.0 * 2 = 0.0
1 = 2 * 0 + 1	.0 * 2 = 0.0
	.0 * 2 = 0.0

Converting to Floating Point

- Step 2: Normalize

$$100100.1001_2 = 1.001001001_2 \times 2^5$$

Converting to Floating Point

- Step 3: Determine S, E, and M

$$\underbrace{+1}_{S}.\underbrace{001001001}_{M}{}_2 \times 2^{\underbrace{5}_{n}}$$

$$
\begin{aligned}
E &= n + 127 \\
&= 5 + 127 \\
&= 132 \\
&= 10000100_2
\end{aligned}
$$

S = 0 (because the value is positive)

Converting to Floating Point

- Step 4: Put S, E, and M together to form 32-bit binary result

$$\underbrace{0}_{S}\ \underbrace{10000100}_{E}\ \underbrace{00100100100000000000000}_{M}{}_2$$

Special Values

Exponent	**Significand**	**Value**
0	0	0
0	nonzero	denormalized number
$1..e_{max}-1$	anything	normal floating point number
e_{max}	0	infinity
e_{max}	nonzero	Not a Number (NaN)

- Single Precision: Exponents of 0 and 255 have special meaning
  - E=0, M=0 represents 0 (sign bit still used so there is +/-0)
  - E=0, M≠0 is a denormalised number ($+/-0.M \times 2^{-126}$) (smaller than the smallest normalised number)
  - E=255, M=0 represents +/- infinity
  - E=255, M ≠ 0 represents NaN (not a number, e.g., returned for 0/0 or sqrt(-1))

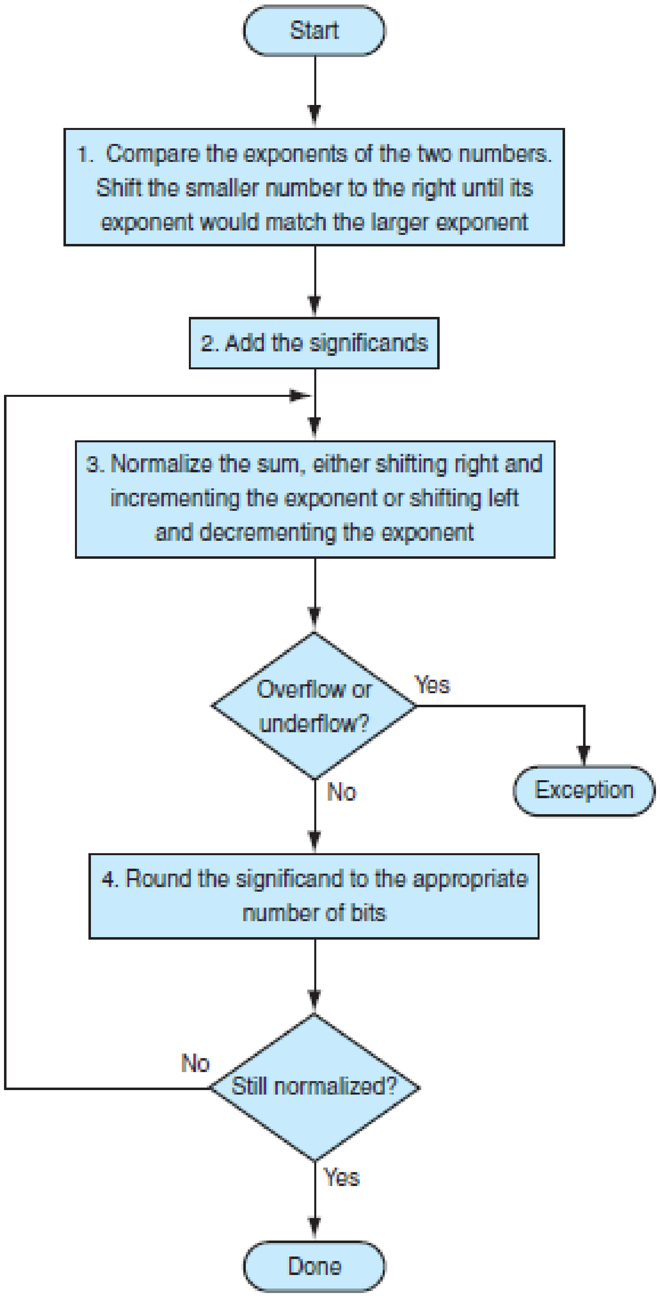
Floating Point Operations

- Arithmetic:
  - multiplication, division:
    - multiply/divide mantissa
    - add/subtract exponent
    - example: $5.6 \times 10^{11} \times 6.7 \times 10^{12} = 5.6 \times 6.7 \times 10^{23}$
  - Addition, subtraction
    - convert operands to have the same exponent value
    - add/subtract mantissas
    - example: $2.1 \times 10^{3} + 4.3 \times 10^{4} = 0.21 \times 10^{4} + 4.3 \times 10^{4}$

Basic Addition Algorithm

1. Align binary points (denormalize smaller number)
    a. compute Diff = Exp(Y) – Exp (X);
    b. Sig(X) = Sig(X) >> Diff
    c. Exp = Exp(Y)
2. Add the aligned components
    - Sig = Sig (x) + Sig (Y)
3. Normalize the sum
    - Shift Sig right/left until leading bit is 1; decrementing or incrementing Exp.
    - Check for overflow in Exp
    - Round (needs more bits, as we will see)
    - repeat step 3 if not still normalized

Basic Addition Algorithm



Addition Example

11.0 + 6.0, 4-bit mantissa

$1.0110 \times 2^3 + 1.1000 \times 2^2$

1. Align binary points (denormalize smaller number)

$$\begin{array}{r} 1.0110 \times 2^3 \\ +\underline{0.1100 \times 2^3} \end{array}$$

2. Add the aligned components

10.0010×2^3 (=17)

3. Normalize the sum

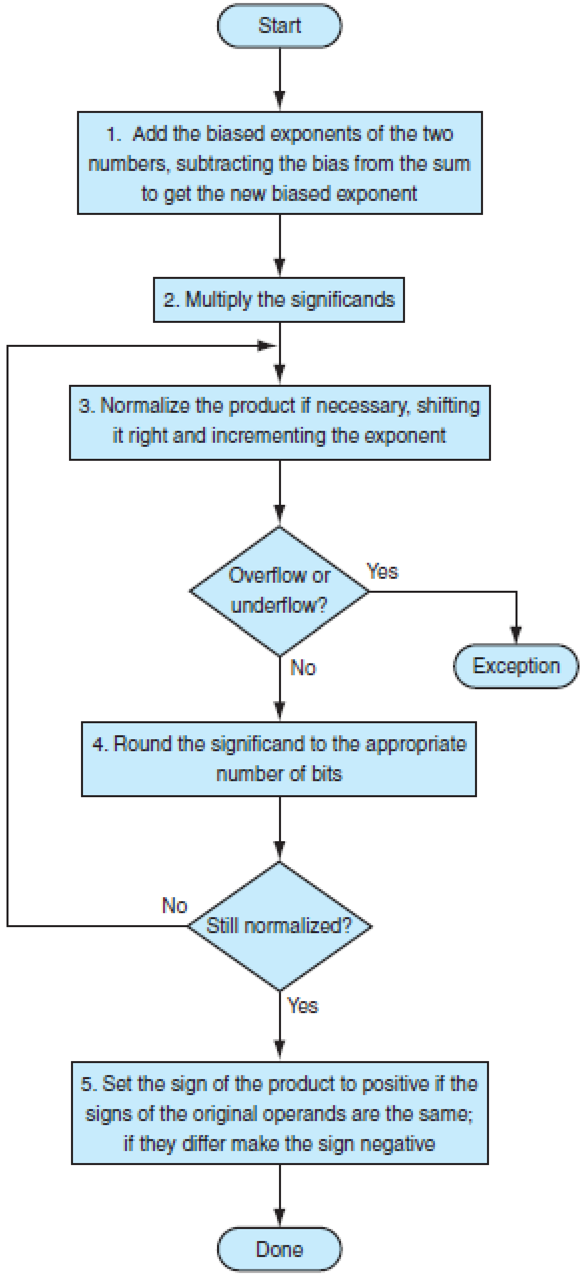
1.0001×2^4

- No overflow, no rounding

Basic Multiplication Algorithm

1. Compute exponents
   - Multiplication: Exp = Exp (X) + Exp (Y) – bias;
   - Division: Exp = Exp (X) – Exp(Y) + bias;
2. Multiply/Divide significands
   - Multiplication: Sig = Sig (X) x Sig (Y);
   - Division: Sig = Sig(X) / Sig(Y);
3. Normalize the product
   - Shift Sig right until leading bit is 1; incrementing Exp.
   - Check for overflow in Exp
   - repeat step 3 if not still normalized
4. Round
   - Any bits that do not fit must be discarded
5. Set sign
   - positive if signs same; negative if signs differ

Basic Multiplication Algorithm



Multiplication Example

.5 * -.4375, 4-bit mantissa

$1.0000_{two} * 2^{-1} * -1.1100_{two} * 2^{-2}$

1. Compute exponents

   $-1 + (-2) = -3$ With Bias: $126 + 125 - 127 = 124$

2. Multiply/Divide significands

   0111000000

3. Normalize the product

   $1.11000000 * 2^{-3}$

4. Round

   $1.1100 * 2^{-3}$

5. Set sign

   $-1.1100 * 2^{-3}$ because original signs differ

Multiplication Example

.5 * -.4375, 4-bit mantissa

$1.0000_{two} * 2^{-1} * -1.1100_{two} * 2^{-2}$

1. Compute exponents

   -1 + (-2) = -3

2. Multiply/Divide significands

   0111000000

3. Normalize the product

   $1.11000000 * 2^{-3}$

4. Round

   $1.1100 * 2^{-3}$

5. Set sign

   $-1.1100 * 2^{-3}$ because original signs differ

```
        11100
      x 10000
  -----------
        00000
       00000
      00000
     00000
    11100
  -----------
    111000000
```

Accuracy and Rounding

- Floating-point numbers are approximations for a number they can't really represent.
  - Infinite possible real numbers between 0 and 1
  - We can only represent 2^{53} of them
  - Approximate by rounding

Rounding Modes

- IEEE Standard has five rounding modes:
  - round to nearest, ties to even (default)
  - round to nearest, ties away from zero
  - round towards plus infinity
  - round towards minus infinity
  - round towards 0

Rounding Hardware

- To round accurately, we need the hardware to include extra bits for the calculation.
- Specifically, we keep 2 extra bits on the right
  - Guard bit
  - Round bit

Guard Bit

- The first bit to the right: an additional digit (bit) used in intermediate calculations to prevent loss of accuracy.

Example for Guard Bit

8.5 – 3.75 = 4.75, 4-bit mantissa

$1.0001 \times 2^3 - 1.1110 \times 2^1$

1. Align binary point:

$$\begin{array}{r} 1.0001 \times 2^3 \\ -\underline{0.0111 \times 2^3} \end{array}$$

2. Subtract the aligned components:

0.1010×2^3

3. Normalize:

1.0100×2^2

Note our answer is actually 5. With only 4-bits we are losing accuracy. Our result would be off by 0.25 or a whole bit in the least significant place.

Example for Guard Bit

8.5 – 3.75 = 4.75, 4-bit mantissa

$1.0001 \times 2^3 - 1.1110 \times 2^1$

1. Align binary point:

$$\begin{array}{rl} 1.0001 & \times\ 2^3 \\ -\underline{0.0111}1 & \times\ 2^3 \\ g & \end{array}$$

2. Subtract the aligned components:

$$\begin{array}{rl} 0.10011 & \times\ 2^3 \\ g & \end{array}$$

3. Normalize:

1.0011×2^2

Now our normalized value is accurate
$1.0011 \times 2^2 = 4.75$

Round Bit

- Bit to the right of guard bit needed for accurate rounding.

Example for Round Bit

- Example: $1.0000 \times 2^{0} - 1.0001 \times 2^{-2}$
  - guard and round bits shown

$$
\begin{array}{rll}
 & 1.0000 \times 2^{0} & \\
- & \underline{0.010001 \times 2^{0}} & \\
 & 0.101111 \times 2^{0} & \text{Result} \\
 & 1.01111 \times 2^{-1} & \text{Normalize} \\
 & 1.1000 \times 2^{-1} & \text{Round; simple round up}
\end{array}
$$

  - Without round bit, result is 1.0111

Sticky Bit

- Round to nearest problems
  - need to know if actual result is closer to the next rounded value up or the next rounded value down.
  - With 4-bit significand, a result of 1.11011 could round to 1.1101 if rounding down or 1.1110 if rounding up
  - Potentially need a much greater number of bits
- Instead keep “sticky” bit (S):
  - used to determine whether there are any 1 bits truncated below the guard and round bits
  - S=1 if any bits are off to the right, otherwise S=0

Example for Sticky Bit

- $1.0000 \times 2^{0} + 1.0001 \times 2^{-5}$
  - guard, round, and sticky bits shown

$$
\begin{array}{rll}
 & 1.0000 \times 2^{0} & \\
+ & \underline{0.0000\underline{1}\underline{0} \times 2^{0}} & 1 \\
 & 1.00001\underline{0} \times 2^{0} & 1
\end{array}
$$

Result

1.0001×2^{0}

Round to nearest
Without S rounds to 1.0000.

Exceptions

- Invalid operation
  - result of operation is a NaN (except = or !=)
  - inf. +/- inf.; 0 * inf; 0/0; inf./inf.; x remainder y, y = 0;
  - sqrt(x) where x < 0, x = +/- inf.
- Overflow
  - result of operation is larger than largest representable number
  - flushed to +/- inf. if overflow exception is not enabled

Exceptions

- Divide by 0
  - x/0 where x = 0, +/- inf.;
  - flushed to +/- inf. if divide by zero exception not enabled
- Underflow
  - subnormal result OR non-zero result underflows to 0
- Inexact
  - rounded result not the actual result (rounding error = 0)

Exceptions

- IEEE Standard specifies defaults and allows traps to permit exceptions to be handled at the program level
  - contrast with the more usual result of aborting the computation altogether.