Floating Point Multipliers:
Simulation & Synthesis Using VHDL

By:

Raj Kumar Singh - B.E. (Hons.) Electrical & Electronics
Shivananda Reddy - B.E. (Hons.) Electrical & Electronics

BITS, PILANI
Outline

- Introduction
  - Multipliers
  - VHDL & Design Flow
- Various Architectures (Multipliers)
  - Simulation
  - Synthesis
  - Analysis
- Conclusion
Real Numbers

- Numbers with fractions
  - 3/5, 4/7
- Pure binary
  - $1001.1010 = 2^4 + 2^0 + 2^{-1} + 2^{-3} = 9.625$
- Fixed point
  - Very limited
- Moving or floating point
  - (almost universal)
  - Widely used in computations
Which base do we use?

- **Decimal:** great for humans, especially when doing arithmetic

- **Hex:** if human looking at long strings of binary numbers, it's much easier to convert to hex and look 4 bits/symbol
  - Not good for arithmetic on paper

- **Binary:** what computers use; computers do +, -, *, / using this only
  - To a computer, numbers always binary
  - Regardless of how number is written:
    
    \[
    32_{\text{ten}} = 32_{10} = 0x20 = 100000_2 = 0b100000
    \]
Floating Point: Overview

- Floating point representation
  - Normalization
  - Overflow, underflow
  - Rounding
- Floating point addition
- Floating point multiply
Floating Point (IEEE-754)

- use a fixed number of bits
  - Sign bit $S$, exponent $E$, significand $F$
  - Value: $(-1)^S \times F \times 2^E$

IEEE 754 standard

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Exponent</th>
<th>Significand</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single precision</td>
<td>32b</td>
<td>8b</td>
<td>23b</td>
<td>$2 \times 10^{+/-38}$</td>
</tr>
<tr>
<td>Double precision</td>
<td>64b</td>
<td>11b</td>
<td>52b</td>
<td>$2 \times 10^{+/-308}$</td>
</tr>
</tbody>
</table>
Normalization

FP numbers are usually normalized
i.e. exponent is adjusted so that leading bit (MSB) of mantissa is 1

Example - Scientific notation where numbers are normalized to give a single digit before the decimal point

\[ \text{e.g. } 3.123 \times 10^3 \]

Because it is always 1, there is no need to store it
FP Overflow / Underflow

- FP Overflow
  - Analogous to integer overflow
  - Result is too big to represent
- FP Overflow
  - Result is too small to represent
  - Means exponent is too small (too negative)
- Both raise Problems, thus need extra Care on their Occurrences in IEEE754
FP Rounding

- Rounding is important
  - Small errors can save the huge storage
- FP rounding hardware helps
  - Finally, keep sticky bit that is set whenever ‘1’ bits are “lost” to the right
    - Differentiates between 0.5 and 0.500000000001

So the rounding can save a huge Memory, of course the price is Accuracy, But that can be paid
Base 2 : Representation

- Number Base B → B symbols per digit:
  - Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Base 2 (Binary): 0, 1

- Number representation:
  - \( d_{31}d_{30} \ldots d_1d_0 \) is a 32 digit number
  - value = \( d_{31} \times B^{31} + d_{30} \times B^{30} + \ldots + d_1 \times B^1 + d_0 \times B^0 \)

- Binary: 0,1 (In binary digits called “bits”)
  - \( 0b11010 \) = \( 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \)
  - = 16 + 8 + 2
  - = 26

#s often written

0b... Here 5 digit binary # turns into a 2 digit decimal #
And in Conclusion...

- We represent “things” in computers as particular bit patterns: \( N \text{ bits} \Rightarrow 2^N \)
- 1’s complement - mostly abandoned
- 2’s complement - universal in computing:
- Overflow: numbers are \( \infty \); computers having finite storage locations, so errors!
VHDL Language

- Hardware Description Language (HDL)
  - High-level language for to model, simulate, and synthesize digital circuits and systems.

- History
  - 1980: US Department of Defense Very High Speed Integrated Circuit program (VHSIC)
  - 1987: Institute of Electrical and Electronics Engineers ratifies IEEE Standard 1076 (VHDL’87)
  - 1993: VHDL language was revised and updated

- Verilog is the other major HDL
  - Syntax similar to C language
The VHDL Language

V  Very High Speed Integrated Circuit

H  Hardware

D  Description

L  Language

• Interoperability between design tools: standardized portable model of electronic systems
• Technology independent description
• Reuse of components described in VHDL
Design Cycle: Simulation

- Functional simulation:
  - simulate independent of FPGA type
  - no timing

- Timing simulation:
  - simulate after place and routing also (back-annotation part)
  - detailed timing
Terminology

- **Behavioral modeling**
  - Describes the functionality of a component/system
  - For the purpose of simulation and synthesis

- **Structural modeling**
  - A component is described by the interconnection of lower level components/primitives
  - For the purpose of synthesis and simulation

- **Synthesis:**
  - Translating the HDL code into a circuit, which is then optimized

- **Register Transfer Level (RTL):**
  - Type of behavioral model used for instance for synthesis
RTL Synthesis

- Input is RTL code
- Compilation & translation
  - Generates technology independent netlist
  - RTL schematic (HDL code analysis)
- Technology mapping
  - Mapping to technology specific structures:
    - Look-up tables (LUT)
    - Registers
    - RAM/ROM
    - DSP blocks
    - Other device specific components/features
- Logic optimization
  - Implementation analysis (technology view)
Most digital systems can be described based on a few basic circuit elements:

- Combinational Logic Gates:
  - NOT, OR, AND
  - Flip Flop
  - Latch
  - Tri-state Buffer

Each circuit primitive can be described in VHDL and used as the basis for describing more complex circuits.
What is an SOC?

- System-on-a-chip, System LSI, System-on-Silicon,
  - Hardware
    - Analog: ADC, DAC, PLL, Tx, Rx, RF Devices
    - Digital: Processor, Memory, Interface, Accelerator, Multiplier, Adder etc…
  - Software
    - OS
    - Application

What are the differences from an ASIC?
Traditional ASIC Design Flow

- Specification Development
- RTL Code Development
- Functional Verification (Simulation)
- Floor-planning, Synthesis, DFT
- Fault Coverage Analysis
- Timing Verification
- Floor-planning, Placement and Route
- Prototyping, Testing, and Characterization
Functional Verification Models

Levels

- Functional
- Behavioral
- RTL
- Logic
- Gate
- Switch
- Circuit
Normally, the accumulator has logical and arithmetic shift capability, both left and right.
Symbol for ALU

ALU operation

a

b

ALU

Zero

Result

Overflow

CarryOut
FP Arithmetic \( x / \div \) (Steps)

- Check for zero, operands
- Add/subtract exponents
- Multiply/divide significands
  - watch sign
- Normalize
- Round
- Double length intermediate results
FP Multiplication: Steps

- Compute sign, exponent, significand
- Normalize
  - Shift left, right by 1
- Check for overflow, underflow
- Round
- Normalize again (if necessary)
FP Multiplication: operations

- **Sign:** \( P_s = A_s \ xor \ B_s \)
- **Exponent:** \( P_E = A_E + B_E \)
  - Due to bias/excess, must subtract bias
    \[
    e = e_1 + e_2 \\
    E = e + 1023 = e_1 + e_2 + 1023 \\
    E = (E_1 - 1023) + (E_2 - 1023) + 1023 \\
    E = E_1 + E_2 - 1023
    \]
- **Significand:** \( P_F = A_F \times B_F \)
  - Standard integer multiply (23b or 52b + g/r/s bits)
  - Use Wallace tree of CSAs to sum partial products
Efficient Multiplier Design

- **Radix-4 Booth Encoding**
  - Used to generate all partial products.

- **Sign Extension Prevention**
  - To prevent sign extension while doing signed number addition (Padding of 1’s).

- **Optimized Wallace Addition Tree**
  - To sum up all operands to 2 vectors (sum, carry).
Multiplier
flowchart

1. Test
Product0 = 1
Product0 = 0

1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register

2. Shift the Product register right 1 bit

32nd repetition?

No: < 32 repetitions
Yes: 32 repetitions

Done

10000
x 1001
10000
00000
00000
1000
1001000

10000
10001000
Step By Step Analysis

Multiply Operation Review

multiplicand (D)
multiplier (Q)

partial product array
(note: can be formed in parallel)

double precision product
(P = Q*D)
Multiply Operation

\[ h = 6 \]

multiplicand (D)
multiplier (Q)

partial product array

double precision product \( P = Q \times D \)
MULTIPLY (unsigned)

- Paper and pencil example (unsigned):

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>1001</td>
</tr>
<tr>
<td></td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>Product</td>
<td>01001000</td>
</tr>
</tbody>
</table>

- **m bits x n bits = m+n bit product**

- **Binary makes it easy:**
  - 0 => place 0 (0 x multiplicand)
  - 1 => place a copy (1 x multiplicand)
  - successive refinement
1. Simultaneous Multiplication

<table>
<thead>
<tr>
<th></th>
<th>X2</th>
<th>X1</th>
<th>X0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \begin{array}{ccc}
X2*Y0 & X1*Y0 & X0*Y0 \\
X2*Y1 & X1*Y1 & X0*Y1 \\
X2*Y2 & X1*Y2 & X0*Y2 \\
\end{array} \]

<table>
<thead>
<tr>
<th>P4</th>
<th>P3</th>
<th>P2</th>
<th>P1</th>
<th>P0</th>
</tr>
</thead>
</table>
Multiplier Schematic: Hardware

How would we develop this logic?
Multiplying Negative Numbers

- This does not work when numbers are negative, then for
- Solution
  Convert to positive if required
  Multiply as above
  If signs were different, negate answer
  Use Booth’s algorithm
Booth’s Algorithm

- Designed to improve speed by using fewer adds
- Works best on strings of 1’s
- Example premise
  - $7 = 8 - 1$
  - $0111 = 1000 - 0001$ (3 adds vs 1 add – 1 sub)
- Algorithm modified to allow for multiplication with negative numbers
Booth’s Encoding

- Really just a new way to encode numbers
  - Normally positionally weighted as $2^n$
  - With Booth, each position has a sign bit
  - Can be extended to multiple bits

<table>
<thead>
<tr>
<th>0-&gt;</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>1-bit Booth</td>
</tr>
<tr>
<td>+2</td>
<td>-2</td>
<td></td>
<td></td>
<td>2-bit Booth</td>
</tr>
</tbody>
</table>
Booth’s Algorithm

<table>
<thead>
<tr>
<th>Current bit</th>
<th>Bit to right</th>
<th>Explanation</th>
<th>Example</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Begins run of ‘1’</td>
<td>00001111000</td>
<td>Subtract</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Middle of run of ‘1’</td>
<td>00001111000</td>
<td>Nothing</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>End of a run of ‘1’</td>
<td>00001111000</td>
<td>Add</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Middle of a run of ‘0’</td>
<td>00001111000</td>
<td>Nothing</td>
</tr>
</tbody>
</table>
## Comparison between various Architectures

<table>
<thead>
<tr>
<th>S. No</th>
<th>Algorithms→</th>
<th>Serial Multiplier (Sequential)</th>
<th>Booth Multiplier</th>
<th>Combination Multiplier</th>
<th>Wallace Tree Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Optimum Area</td>
<td>110 LUTs</td>
<td>134 LUTs</td>
<td>4 LUTs</td>
<td>16 LUTs</td>
</tr>
<tr>
<td>2.</td>
<td>Optimum Delay</td>
<td>9 ns</td>
<td>11 ns</td>
<td>9 ns</td>
<td>9 ns</td>
</tr>
<tr>
<td>3.</td>
<td>Sequential Elements</td>
<td>105 DFFs</td>
<td>103 DFFs</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>4.</td>
<td>Input/Output Ports</td>
<td>67 / 71</td>
<td>50 / 49</td>
<td>4 / 4</td>
<td>24 / 18</td>
</tr>
<tr>
<td>5.</td>
<td>CLB Slices(%)</td>
<td>57(7.42%)</td>
<td>71(36.98%)</td>
<td>2(1.04%)</td>
<td>8(4.17%)</td>
</tr>
<tr>
<td>6.</td>
<td>Function Generators</td>
<td>114(7.42%)</td>
<td>141(36.72%)</td>
<td>4(1.04%)</td>
<td>16(4.17%)</td>
</tr>
<tr>
<td>7.</td>
<td>Data Required Time/Arrival Time</td>
<td>9.54 ns, 8.66 ns</td>
<td>9.54 ns, 9.36 ns</td>
<td>NA, 8.61</td>
<td>10 ns, 8.52 ns</td>
</tr>
<tr>
<td>8.</td>
<td>Optimum Clock (MHz)</td>
<td>100</td>
<td>101.9</td>
<td>NA</td>
<td>100</td>
</tr>
<tr>
<td>9.</td>
<td>Slack</td>
<td>0.89 ns</td>
<td>0.19 ns</td>
<td>Unconstrained path</td>
<td>1.48 ns</td>
</tr>
</tbody>
</table>
Observations on Multiplication

- Can speed up algorithm by doing 2 bits at a time, instead of just one
  - Using Booth encoding strategy (in more depth)

Multiplication algorithm

- Sequential version are more efficient than combinational in terms of Hardware, Synchronization, speed
- Can use carry save adders instead of ripple adder
- A Wallace tree structure to combine the partial products is another excellent enhancement in Architecture
Suppose there are two numbers M, N. We have to find $A = M \times N$, let's assume the 
M & N both are B base number And also $M < N$.

$$A = MN - (M \times B - N \times (M-1))$$

**Next step:** Subtract the $M \times B$ from $MN$, Where $MN$ can be found by just writing both the numbers into a large register, And $M \times B$ is also easy to generate.

It is just shifting towards left of operand with zero padding. Again we will restore the number $(M-1)$ in place of $M$ by just decrementing.

The continuous iteration will decrement the $M$ and finally it will reach to 1.
Q and A