Area Efficient Floating-Point Adder And Multiplier with IEEE-754 Compatible Semantics

Andreas Ehliar <andreas.ehliar@liu.se>
Linköping University

2014-12-11

Presented at FPT2014
Given a floating point heavy reference code written in C, Java, Python, etc, how should we port this application to an FPGA?
Porting reference code to FPGAs

- Given a floating point heavy reference code written in C, Java, Python, etc, how should we port this application to an FPGA?
  - **Rewrite the algorithms to use fixed-point math**
    - Will probably give the most efficient implementation
    - Will probably require the most porting effort to verify that the numerical accuracy is adequate
Porting reference code to FPGAs

- Given a floating point heavy reference code written in C, Java, Python, etc, how should we port this application to an FPGA?
- Rewrite the algorithms to use fixed-point math
- **Rewrite the algorithms to use a floating point operators better suited for FPGAs than IEEE-754**
  - Example 1: 17 bit mantissa instead of 24
  - Example 2: Remove support for subnormal numbers
- Will still require some effort to verify that the numerical accuracy is adequate
Porting reference code to FPGAs

Given a floating point heavy reference code written in C, Java, Python, etc, how should we port this application to an FPGA?

- Rewrite the algorithms to use fixed-point math
- Rewrite the algorithms to use a floating point operators better suited for FPGAs than IEEE-754
- **Use floating point operators that are fully IEEE-754 compliant**
  - Disadvantage: Impact on area, power, and $f_{\text{max}}$
  - Advantage: No verification of the numerical accuracy is required. Reference algorithms can be moved directly to hardware.
My Proposal

- Use a number format which is more suitable for efficient floating point operators in FPGAs than the 32 bit (or 64 bit) IEEE-754 format
- Make sure that the operators nevertheless give the same numerical result as mandated by IEEE-754
- The cost of converting between the custom number format and IEEE-754 can be amortized over several floating point operations.

Andreas Ehliar<br>andreas.ehliar@liu.se<br>Linköping University

Area Efficient Floating-Point Adder And Multiplier with IEEE-754
High-Radix Floating Point Numbers

Proposal: Use a higher radix to enable coarser shifts

Normal floating point: \((-1)^{\text{sign}} \times \text{mantissa} \times 2^{\text{exponent}}\)
In this work: \((-1)^{\text{sign}} \times \text{mantissa} \times 16^{\text{exponent}}\)

Historical note: Radix 16 was used in some IBM mainframes (System/360 and up)
### HRFP\textsubscript{16} format

<table>
<thead>
<tr>
<th>Format type</th>
<th>Bit number</th>
<th>Field name</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE-754 (single precision)</td>
<td>31</td>
<td>Sign bit</td>
</tr>
<tr>
<td></td>
<td>30-23</td>
<td>Exponent</td>
</tr>
<tr>
<td></td>
<td>22-0</td>
<td>Mantissa (with implicit one)</td>
</tr>
<tr>
<td>HRFP\textsubscript{16} (this paper)</td>
<td>35</td>
<td>Sign bit</td>
</tr>
<tr>
<td></td>
<td>34-33</td>
<td>If 10: \textit{Inf}.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If 11: \textit{NaN}.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If 00 or 01: Normal value</td>
</tr>
<tr>
<td></td>
<td>33-27</td>
<td>Exponent</td>
</tr>
<tr>
<td></td>
<td>26-0</td>
<td>Mantissa (with explicit one)</td>
</tr>
</tbody>
</table>

- Note: 36 matches the width of the BlockRAM
### HRFP\textsubscript{16} format

<table>
<thead>
<tr>
<th>Format type</th>
<th>Bit number</th>
<th>Field name</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE-754 (single precision)</td>
<td>31</td>
<td>Sign bit</td>
</tr>
<tr>
<td></td>
<td>30-23</td>
<td>Exponent</td>
</tr>
<tr>
<td></td>
<td>22-0</td>
<td>Mantissa (with implicit one)</td>
</tr>
<tr>
<td>HRFP\textsubscript{16} (this paper)</td>
<td>35</td>
<td>Sign bit</td>
</tr>
<tr>
<td></td>
<td>34-33</td>
<td>If 10: \textit{Inf}.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If 11: \textit{NaN}.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If 00 or 01: Normal value</td>
</tr>
<tr>
<td></td>
<td>33-27</td>
<td>Exponent</td>
</tr>
<tr>
<td></td>
<td>26-0</td>
<td>Mantissa (with explicit one)</td>
</tr>
</tbody>
</table>

- Problem: HRFP\textsubscript{16} can represent more values than single-precision IEEE754
### HRFP<sub>16</sub> format

<table>
<thead>
<tr>
<th>Format type</th>
<th>Bit number</th>
<th>Field name</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE-754 (single precision)</td>
<td>31</td>
<td>Sign bit</td>
</tr>
<tr>
<td></td>
<td>30-23</td>
<td>Exponent</td>
</tr>
<tr>
<td></td>
<td>22-0</td>
<td>Mantissa (with implicit one)</td>
</tr>
<tr>
<td>HRFP&lt;sub&gt;16&lt;/sub&gt; (this paper)</td>
<td>35</td>
<td>Sign bit</td>
</tr>
<tr>
<td></td>
<td>34-33</td>
<td>If 10: <em>Inf.</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>If 11: <em>NaN.</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>If 00 or 01: Normal value</td>
</tr>
<tr>
<td></td>
<td>33-27</td>
<td>Exponent</td>
</tr>
<tr>
<td></td>
<td>26-0</td>
<td>Mantissa (with explicit one)</td>
</tr>
</tbody>
</table>

- Solution: HRFP<sub>16</sub> restricts the exponent and mantissa so that only values legal in IEEE-754 are allowed.
Hardware overview

Floating Point Adder
- Compare
- Swap
- Align
- Add
- Normalize
- Round/Postprocess

Floating Point Multiplier
- Multiply
- Normalize
- Round/Postprocess
Adder - Compare stage

- Compares the magnitude of the input operands.
- Compares for ≤ or < based on sign bit
  - LUT6_2 were manually instantiated to do this optimally

Area: 18 LUTs, 73 registers
Adder - Swap

- Swaps the inputs (if required)
  - Synthesis tool neatly manages to exploit both outputs of the LUT6_2 here
- Also calculates the exponent difference

Area: 48 LUTs, 68 registers
Adder - Align

- Shifts mantissa in steps of four bits at a time
  - Key difference between HRFP\textsubscript{16} and IEEE-754 operators
- Implemented as an 8-to-1 multiplexer
  - Split into two 4-to-1 multiplexers combined with a 2-to-1 multiplexer in next stage
- This stage is also responsible for sticky bit generation

Area: 35 LUTs, 70 registers

Compare
Swap
Align
Add
Normalize
Round/Postprocess

Andreas Ehliar <andreas.ehliar@liu.se> Linköping University
Area Efficient Floating-Point Adder And Multiplier with IEEE-754 Compatible Semantics
Adder - Add

- Adds or subtracts the two mantissae together
- Also contains the final 2-to-1 multiplexer of the alignment step
  - Merged into the same LUTs as the adder/subtractor

Area: 32 LUTs, 40 registers
LZD (Leading zero detection) and shifter
Operates on four bits at a time
- Key difference between HRFP_{16} and IEEE-754 operators
The LZD consists of manually instantiated LUTs (to enable RLOC-based floorplanning)
Normalization/denormalization is a problem in FPGA implementations:

- Example: $\text{NORMALIZE}(0.0010001) = 1.0001000$
- Leading zero detection combined with large shifter
The value of the four MSB bits decides how rounding should happen.

- Key difference between HRFP\textsubscript{16} and IEEE-754 operators

Area: 50 LUTs, 0 registers
Mantissa post-processing

- Allowed mantissa values:
  - $1\ldots000$
  - $01\ldots00$
  - $001\ldots0$
  - $0001\ldots$
  - $000000\ldots$ (special case)
Rounding

- Rounding in IEEE-754:
  - 1xxx xxxx xxxx xxxx xxxx xxxx GRS

- Rounding in HRFP<sub>16</sub>
  - 1xxx xxxx xxxx xxxx xxxx xxxx GRSs ss
  - 01xx xxxx xxxx xxxx xxxx xxxx xGRS ss
  - 001x xxxx xxxx xxxx xxxx xxxx xxGR Ss
  - 0001 xxxx xxxx xxxx xxxx xxxx xxxxG RS

- G: Guard bit, R: Round bit, S: Sticky bit, s: Bits that contribute to the sticky bit.

- (GRSs are all set to 0 in the final mantissa)
Multiplier - Multiply

- Contains a $27 \times 27$ bit multiplier

Area: 31 LUTs, 57 registers, 2 DSP48E1

Andreas Ehliar <andreas.ehliar@liu.se> Linköping University

Area Efficient Floating-Point Adder And Multiplier with IEEE-754 Compatible Semantics
Little overhead in FPGAs with $18 \times 18$ multipliers (requires 4 DSP blocks unless Karatsuba-like algorithms are used)

Newer Xilinx FPGAs use $25 \times 18$ bit multipliers however

Possible solution: Create a $27 \times 24$ bit multiplier in two DSP blocks and use LUTs for the remaining $27 \times 3$ bit multiplier.
27 × 27 bit multiplication

- My solution: Use the pre-adder of the DSP48E1 block combined with an adder as shown below:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>$A \times B = 8 \times A[26:3] \times B$</td>
<td>$+$</td>
<td>$(0 \ + \ 0)$</td>
</tr>
<tr>
<td>001</td>
<td>$A \times B = 8 \times A[26:3] \times B$</td>
<td>$+$</td>
<td>$(B \ + \ 0)$</td>
</tr>
<tr>
<td>010</td>
<td>$A \times B = 8 \times A[26:3] \times B$</td>
<td>$+$</td>
<td>$(0 \ + \ 2B)$</td>
</tr>
<tr>
<td>011</td>
<td>$A \times B = 8 \times A[26:3] \times B$</td>
<td>$+$</td>
<td>$(B \ + \ 2B)$</td>
</tr>
<tr>
<td>100</td>
<td>$A \times B = 8 \times (A[26:3] + 1) \times B$</td>
<td>$-$</td>
<td>$(2B \ + \ 2B)$</td>
</tr>
<tr>
<td>101</td>
<td>$A \times B = 8 \times (A[26:3] + 1) \times B$</td>
<td>$-$</td>
<td>$(B \ + \ 2B)$</td>
</tr>
<tr>
<td>110</td>
<td>$A \times B = 8 \times (A[26:3] + 1) \times B$</td>
<td>$-$</td>
<td>$(0 \ + \ 2B)$</td>
</tr>
<tr>
<td>111</td>
<td>$A \times B = 8 \times (A[26:3] + 1) \times B$</td>
<td>$-$</td>
<td>$(B \ + \ 0)$</td>
</tr>
</tbody>
</table>

- Warning: This is not a general 27 × 27 bit multiplier: If the A input is all ones the pre-added one will cause an overflow.
  - However, the mantissa in HRFP$_{16}$ never consists of only ones
Normalization based on the four MSB bits

- Key difference between HRFP\textsubscript{16} and IEEE-754 operators
Rounding is handled similarly to the adder except that IEEE-754 style gradual underflow is also emulated.

- **Key difference between HRFP\textsubscript{16} and IEEE-754 operators**
- Manually instantiated flip-flops for final register to enable efficient use of SR-input

Area: 204 LUTs, 277 registers
Corner case in IEEE-754

- Normally, the MSB bit of the mantissa is always 1 (e.g.: \(1.xxxxx \times 2^{\text{exponent}}\))
- However, if bit 30-23 is zero: the MSB bit is allowed to be zero. (This is often called a subnormal or denormalized number)
Traditional wisdom in FPGA community is that gradual underflow support is too costly
  - Requires a very large normalization stage.

If you actually encounter numbers in this range, just add an extra exponent bit instead.
An extra exponent bit is added and small exponents are handled using a special case as seen below:

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Mantissa before post-processing and rounding</th>
</tr>
</thead>
<tbody>
<tr>
<td>≥ 64</td>
<td>1xxx xxxx xxxx xxxx xxxx xxxx xxxx xGRS ss</td>
</tr>
<tr>
<td>Normal</td>
<td>≥ 64</td>
</tr>
<tr>
<td>numbers</td>
<td>&gt; 64</td>
</tr>
<tr>
<td>subnormal</td>
<td>60</td>
</tr>
<tr>
<td>numbers</td>
<td>59</td>
</tr>
<tr>
<td>≤ 57</td>
<td>¬Normal numbers</td>
</tr>
</tbody>
</table>

Flushed to zero
Verification

- FPGen from IBM
- Home built testing code where all of the following combinations are tested:
  - All sign and exponent combinations
  - All mantissa combinations where only one bit is zero
  - All mantissa combinations where only one bit is non-zero
  - All values from 0x7fff00 to 0x7fffff
  - All values from 0x000000 to 0x000100
  - Some random values
- SoftFloat was used as a Golden Model
Implementation Results (xc7k70t-1-fbg484)

<table>
<thead>
<tr>
<th>Unit</th>
<th>Slice LUTs</th>
<th>Slice Registers</th>
<th>DSP48E1 blocks</th>
<th>$f_{\text{max}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>HRFP$_{16}$ Adder</td>
<td>261</td>
<td>324</td>
<td>0</td>
<td>319 MHz</td>
</tr>
<tr>
<td>HRFP$_{16}$ Multiplier</td>
<td>235</td>
<td>277</td>
<td>2</td>
<td>305 MHz</td>
</tr>
<tr>
<td>Coregen Adder</td>
<td>363</td>
<td>209</td>
<td>0</td>
<td>291 MHz</td>
</tr>
<tr>
<td>Coregen Multiplier</td>
<td>119</td>
<td>76</td>
<td>2</td>
<td>464 MHz</td>
</tr>
</tbody>
</table>

- Note: Coregen multiplier does not support gradual underflow! Adding such support would result in a similar area (if not larger)
Future work

- Double precision IEEE-754
- Add other rounding modes besides round to nearest even
- Hard blocks in FPGAs?
Conclusions

- This kind of operator can be useful if your results should be numerically equal to IEEE-754 and you
  - have more adders/subtracters than multipliers
  - require support for IEEE-754 gradual underflow