TSTE18 Digital Arithmetic
Seminar 9

Oscar Gustafsson

Floating-point arithmetic

- Floating-point representations
- IEEE 754
- Rounding

- **Floating-point addition/subtraction**
- **Floating-point multiplication**
- **Floating-point division**
- **Fused floating-point multiply-add**

Floating-point addition/subtraction

- Assuming binary32 and $E_X \geq E_Y$, it is possible to factor out the exponent term as

$$\begin{aligned} Z &= (-1)^{s_Z} M_Z \, 2^{E_Z - 127} = \\ X \pm Y &= \left((-1)^{s_X} M_X \pm (-1)^{s_Y} M_Y \, 2^{-(E_X - E_Y)}\right) 2^{E_X - 127} \end{aligned}$$

  where we can identify

$$(-1)^{s_Z} \hat{M}_Z = (-1)^{s_X} M_X \pm (-1)^{s_Y} M_Y \, 2^{-(E_X - E_Y)} \quad (1)$$

  and

$$\hat{E}_Z = E_X . \quad (2)$$

- Hence, in general we would like to find

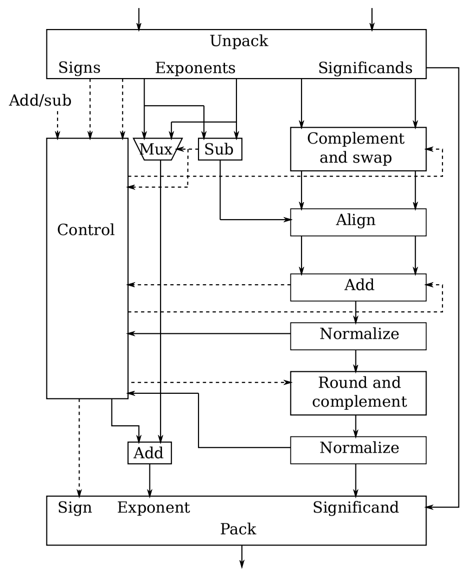
$$(-1)^{s_Z} \hat{M}_Z = (-1)^{s_X} M_X \pm (-1)^{s_Y} M_Y \, 2^{-|E_X - E_Y|} \quad (3)$$

  and

$$\hat{E}_Z = \max\{E_X, E_Y\}. \quad (4)$$

Floating-point addition/subtraction

- Direct realization of a floating-point adder/subtracter



Floating-point addition example

- Consider the addition of the following normalized floating-point numbers

Sign	Exponent	Normalized significand
0	011	00110101
1	001	10011010

Floating-point addition/subtraction

- Aligning: use a programmable shifter with binary input control

Floating-point addition/subtraction

- Leading zeros/ones detection or prediction
- First add and then detect
- Predict in parallel with adding
- Can be separated in two stages:
  - First, make a coarse estimate and use for the coarse shift
  - Then, fine tune the estimate and shift

Floating-point addition/subtraction

- Use two different data paths depending on amount of alignment
- Subtraction with 0 or 1 bit preshift
  - Simple preshift
  - Cancellation may occur, so an arbitrary postshift
- Subtraction with more than one bit preshift and arbitrary preshift addition
  - Arbitrary preshift
  - 0 or 1 bit postshift (no need for leading zeros/ones prediction)
- Beneficial since the arbitrary shift and the adder have the longest delays
- Only two long delay operations in the critical path

Floating-point addition/subtraction

- Must also handle the defined exceptions
- Overflow can only occur with normalizing right-shifts
- Underflow can only occur with normalizing left-shifts
- Zero detection can be handled in the leading 0s detection
- NaN and invalid operations handled in the unpacking and packing blocks

Floating-point multiplication

- The multiplication of two floating-point numbers (assumed to be in IEEE 754 binary32 format) is computed as

$$\begin{aligned} Z &= (-1)^{s_Z}\, M_Z\, 2^{E_Z-127} = \\ XY &= (-1)^{s_X}\, M_X\, 2^{E_X-127}(-1)^{s_Y}\, M_Y\, 2^{E_Y-127} \qquad (5) \end{aligned}$$

where we see that

$$\begin{aligned} s_Z &= s_X \oplus s_Y & (6) \\ \hat{M}_Z &= M_X M_Y & (7) \\ \hat{E}_Z &= E_X + E_Y - 127. & (8) \end{aligned}$$

Floating-point multiplication

- With $1 \leq M_X, M_Y < 2$, we get $1 \leq \hat{M}_Z < 4$
- It may be required to shift $\hat{M}_Z$ one position to the right to obtain the normalized value M_Z
- If this happens one will also need to add 1 to $\hat{E}_Z$ to obtain E_Z
- Rounding depends on the less significant bits while normalization requires the most significant bit
- Hence, it is possible to compute the alternative rounded and normalized results in parallel with determining which normalization should be used
- Overflow or underflow can happen both during multiplication and normalization
- If the exponent overflows we should set the output to be $\pm\infty$
- If the exponent underflows, we can either go for denormalized numbers or set the output to 0

Floating-point multiplication

- A multiplication corresponds to one fixed-point multiplication, one fixed-point addition, and a simple normalizing step after the operations

Floating-point multiplication example

- Consider the multiplication of the following normalized floating-point numbers

Sign	Exponent	Normalized significand
0	011	00110
1	001	10011

Floating-point division

- The division of two floating-point numbers (assumed to be in IEEE 754 binary32 format) is computed as

$$Z = (-1)^{s_Z}\, M_Z\, 2^{E_Z-127} = \frac{X}{Y} = \frac{(-1)^{s_X}\, M_X\, 2^{E_X-127}}{(-1)^{s_Y}\, M_Y\, 2^{E_Y-127}} \tag{9}$$

  where we see that

$$\begin{aligned} s_Z &= s_X \oplus s_Y && (10)\\ \hat{M}_Z &= \frac{M_X}{M_Y} && (11)\\ \hat{E}_Z &= E_X - E_Y + 127. && (12)\end{aligned}$$

Floating-point division

- With $1 \leq M_X, M_Y < 2$, we get $\frac{1}{2} < \hat{M}_Z < 2$
- It may be required to shift $\hat{M}_Z$ one position to the left to obtain the normalized value M_Z
- If this happens one will also need to subtract 1 to $\hat{E}_Z$ to obtain E_Z
- The remainder of the division acts as the sticky bit
- Overflow or underflow can happen both during division and normalization
- If the exponent overflows we should set the output to be $\pm\infty$
- If the exponent underflows, we can either go for denormalized numbers or set the output to 0
- Need to handle divide-by-zero

Floating-point division

- A division corresponds to one fixed-point division, one fixed-point subtraction, and a simple normalizing step after the operations

Fused multiply-add

- The fused-multiply-add (FMA) operation, $P = AX + B$ have some commonly used applications
  - Polynomial evaluation

$$f(z) = c_n z^n + c_{n-1} z^{n-1} + \cdots + c_1 z + c_0 \tag{13}$$

```
s := 0
for j in n downto 0
  s := s*z + c(j)
end
```

  - Sum-of-products

$$f(z) = h(0)x(0) + h(1)x(1) + \cdots + h(n)x(n) \tag{14}$$

```
s := 0
for j in 0 to n
  s := h(j)*x(j) + s
end
```

Fused multiply-add

- Alignment preshift can be done in parallel with the multiplication
- Compute $\underbrace{E_A + E_X}_{\text{Exponent of } AX} - E_B$ and use for alignment
- Leading 0/1 prediction can be done in parallel with the carry-propagation using a redundant output of the multiplier
- No need to round and normalize the intermediate result of the multiplication

Fused multiply-add example

- Compute $AX + B$ using a fused multiply-add

Variable	Sign	Exponent	Normalized significand
A	0	010	00110
X	0	001	10011
B	0	010	01101

Synthesizing custom floating-point data paths

- Normalization and rounding are required to obtain a valid floating-point value to store
- These are at the same time quite time and area consuming operations
- Hence, when synthesizing custom floating-point data paths it is often beneficial to store intermediate results in a non-rounded and non-normalized format
- IEEE 754 defines these extended formats for internal computations of e.g. sin and cos
- Altera has a synthesis tool that can automatically map floating-point algorithms to FPGAs, only normalizing and rounding the required outputs rather than the intermediate results