Fixed and floating point numbers in VHDL

Engineering Department in Ferrara

Motivations

- Machine learning applications such as those based on neural networks require the use of real values
- Floating point units are no longer a need of scientific calculus only
- **VHDL - 2008** supports both fixed and floating point customizable data types
  - why customize?
  - **unuseful bits occupy area, increase delays and dissipate power**
  - bit level accessibility to data
- Format compatible with IEEE-754 floating point standard
- Growing support from simulation and synthesis EDA tools

Trade-offs (VLSI)

- Consider for instance parallel multipliers and dividers
- The delay is proportional to word width ($O(n)$);
- The cost and power dissipation are quadratic in word width ($O(n^2)$);
- Word width reduction may be a problem because of accuracy, precision and numerical stability

Summary

VHDL fixed point package

- Package & types
- Format
- Sizing
- Overloading
- Assignments and expressions

Package & types

- **`use ieee.fixed_pkg.all;`**
- **ufixed** unsigned fixed point
  `type ufixed is array (integer range <>) of std_logic;`
- **sfixed** signed fixed point
  `type sfixed is array (integer range <>) of std_logic;`
- unresolved types

Fixed point format

- **`constant x: ufixed(3 downto -3) := "0100101"`**
- whole number from index 3 downto 0: 0100=4
- fractional number on the right from index -1 downto -3: 101=0.625
- x=4.625
- it is legal to have only an integer (4 downto 0) or a fraction (-1 downto -6)

Definition

`ufixed(k downto -j)` assigned to $i_k i_{k-1} .... i_0 \,.\, f_{-1} f_{-2} .... f_{-j}$ corresponds to the fractional number $i_k 2^k + i_{k-1} 2^{k-1} + ... + f_{-1} 2^{-1} + f_{-2} 2^{-2} + i_0 2^0 .... + f_{-j} 2^{-j}$

Fixed precision math

- Main difference with **`signed`** and unsigned data types: the math is full precision and not in modulo
- The result size must accomodate for all the bits of the result

```
signal a43, b43 : ufixed(3 downto -3);
signal y53: ufixed(4 downto -3);
....
y53 <= a43+b43;
```

- Overloading for **`+,-,*,/,rem,mod,abs,=,/=,>,<,>=,<=`** : ufixed is compatible with **ufixed**, **real** and **natural**

Assignments

- **`signal x:  ufixed (3 downto -3);`**
- to a string literal: **`x <="0110100"; - 6.5`**
- to a real (integer) literal: **`x <= to_ufixed(6.5,a);`** (the base conversion sized to **`a`** that is equivalent to texttt **x <= to_ufixed(6.5,3,-3);**)
- **rules for function matching in VHDL**: search in libraries for a function with the same name of the callee, if more functions are found search for the one with compatible parameters number and types

Expressions

```
signal a43 : ufixed(3 downto -3);
signal y53: ufixed(4 downto -3);
....
y53 <= a43+"0111001"; -- not correct
y53 <= a43+6.9;
y53 <= a43+9;

signal a43 : ufixed(3 downto -3);
signal y63: ufixed(6 downto -3);
....
y63 <= y63+a43; -- not correct, result too large
```

- use the function **`resize`** to constrain the size of the result

```
y63 <= resize(arg=>y63+a43,size_res=>y63,
       overflow_style=>fixed_saturate/fixed_wrap,
       round_style=>fixed_round/fixed_truncate);
```

Fixed point conversions

- **`to_ufixed`** : integer, real, unsigned, std_logic_vector to ufixed
- **`to_sfixed`** : integer, real, unsigned, std_logic_vector to sfixed
- **`resize`** : ufixed to ufixed, sfixed to sfixed
- **`add_sign`** : ufixed to sfixed
- **`to_real`** : ufixed o sfixed to real
- **`to_integer`**: ufixed o sfixed to integer
- **`to_unsigned`**: ufixed to unsigned
- **`to_signed`**: sfixed to signed
- **`to_slv`**: ufixed o sfixed to std_logic_vector

Signed fixed point

- Complement two notation for the integer part
- Positive for the fractional part
- Example **`signal q:  sfixed(5 downto -4) := "1100100011";`**
  - integer part (from 5 to 0) `110010` that is equal to -14
  - fractional part (from -1 to -4) `0011` that is equal to 0.1875
  - q=-13.8125

Signed fixed point

- The manual conversion from base ten is as follows:
  - convert the integer part to its complement two representation
  - if the integer part is ≥ 0 convert the fractional part to binary
  - if the integer part is < 0 and the fractional part is > 0 subtract 1 from the integer part and convert 1 minus the fractional part to binary
  - if the fractional part is 0 simply place 0s

Summary

Fixed point

Floating point

Projects

Floating point numbers

- Allow for a flexible use of the available digits to represent numbers with different orders of magnitudo
- $y = s \times b^e$ where y is the number value, s is the significand, b is the base and e is the exponent
- FPUs are a necessary part of all general purpose CPUs
- IEEE-754 standard
- Up to the last years floating point calculations were mainly performed by general purpose CPUs
- **Machine learning and power/delay constraints are focusing the attention to accelerator architectures performing specific tasks**
- Specific tasks may require **data formats different from the standards used in FPUs**

Floating point type and subtypes

- Library and package: `use ieee.float_pkg.all;`
- `type float is array (integer range <>) of std_logic;`
- The type is customizable, but some predefined subtype is available to emulate the IEEE 754 standards that you have seen in computer architecture
  - **single precision**: `subtype float32 is float(8 downto -23);`
  - **double precision**: `subtype float64 is float(11 downto -52);`
  - **IEEE 854 extended precision**: `subtype float128 is float(15 downto -112);`

Format

- `signal y: float (p downto -q);` where $q \leq -6$
- The leftmost bit (`y(p)`) is used for sign
- The `p` bits from `y(p-1)` to `y(0)` are used for the exponent
- The `q` bits from `y(-1)` to `y(-q)` are used for the (positive) fraction (similar to the significand in the scientific notation)
- The value is biased by $2^{p-1} - 1$. This allows for the use of natural numbers in the exponent

$$v(y) = (-1)^{y(p)} \times 2^{e-(2^{p-1}-1)} \times (1.0 + f)$$

Example: normal numbers

- normal number (`y(-1)=1`) the leading 0s are moved to the exponent
- `signal y: float (4 downto -10):="001001100000000";`
  - positive sign
  - the exponent is given by **0100** which is equal to 4
  - the fraction is $2^{-1} + 2^{-2} = 0.75$
  - $v(y) = 2^{4-2^3+1} \times (1.0 + 0.75) = 2^{-3} \times 1.75 = 0.21875$
- `signal y: float (4 downto -10):="010011001000000";`
  - positive sign
  - the exponent is given by **1010** which is equal to 9
  - the fraction is $2^{-1} + 2^{-4} = 0.5625$
  - $v(y) = 2^{9-2^3+1} \times (1.0 + 0.5625) = 2^2 \times 1.75 = 6.25$

Special cases

- When using the `float` type in the previous slide, the minimal abs value for normal numbers would be $2^{0-7} \times (1.0 + 0.5) = 0.01171875$
- If we remove the hypothesis `y(-1)=1` , we have sub-normal numbers that are smaller than the minimum normal number
- They have the exponent's bits equal to 0 (i.e. they are multiplied by $e^{-(2^{p-1}-1)}$), while fraction's bits are allowed to have any value and are added to 0.0 and not 1.0
- Example: `signal y: float (4 downto -10):=000000100010000;` has value $v(y) = 2^{-7} \times (2^{-2} + 2^{-6}) = 0.002075195$

Special cases

- The IEEE-754 standard defines exceptions for floating point operations that can be used by programs to avoid the propagtion of errors to data outputs
- Conventions for ∞ and *NaN* for `float(8 downto -23);`
- $+\infty$: `0 11111111 00000000000000000000000`
- $-\infty$: `1 11111111 00000000000000000000000`
- *NaN*: `1 11111111 00000000000000000000001`

Parameters

- VHDL allows for the definition of floating point parameters such as the kind of rounding used
- They may significantly affect numerical computations and they result in small modifications in FP arithmetic units
- We will consider default parameters

Operators

- Arithmetic overloading on floats
  - +,-,*,/
  - rem, mod, abs
  - =, /=, >, <, >=, <= (avoid = and /=)
- Floats can be combined with reals and integers still returning a float

Conversion I

- `signal y:  float (4 downto -10);`
- String to float: `y <="000101001001000";`
- Real to float: `y <=to_float(6.5,y);`
- Real to float: `y <=to_float(6.5,4,-10);`
- Uses rounding (try to convert the decimal 0.1 number)
- `to_float` works also with fixed data types, signed, unsigned and `std_logic_vector`

Conversion II

- `resize:` float to float with potential rounding
- `to_real, to_integer, to_sfixed, to_ufixed, to_unsigned, to_signed:` convert floats to the corresponding data types
  - use the simulator to understand the behavior in case of overflow
- `to_slv :` converts floats to `std_logic_vector`

Summary

Fixed point

Floating point

Projects

Data size

- Analysis of the relationship between data size and accuracy
- Function evaluation for e^{-x} and $ln(x)$ via the use of Taylor series which require only adders and multipliers
  - in the nearby of 0.0, $e^{-x} = \sum_{n=0}^{\infty} \frac{(-1)^n x^n}{n!}$
  - in the nearby of 2.0, $ln(x) = ln(2) + \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n2^n}(x-2)^2$
  - in the nearby of 0.0, $\frac{1}{1+e^{-x}} = \frac{1}{2} + \frac{1}{4}x - \frac{1}{48}x^3 + ...$
- Dataflow description whose accuracy has to be compared with that achievable using the `real` data type
- Available for both fixed and floating point

Pseudo-random generation of floating point numbers

- Pseudo-random generation is commonly used in test and built-in self-test
- Compact LFSR based on-chip test generators are available that produce words with a uniform bit distribution
- A uniform distribution in the word's bits does not correspond to a uniform distribution of the corresponding floats
- Problem analysis and implementation