CO405H

Computing in Space with OpenSPL

Topic 11: Numerics I

Oskar Mencer **Georgi Gaydadjiev**

Department of Computing
Imperial College London

http://www.doc.ic.ac.uk/~oskar/
http://www.doc.ic.ac.uk/~georgig/

CO405H course page: http://cc.doc.ic.ac.uk/openspl16/
WebIDE: http://openspl.doc.ic.ac.uk
OpenSPL consortium page: http://www.openspl.org

o.mencer@imperial.ac.uk g.gaydadjiev@imperial.ac.uk

Lecture Overview

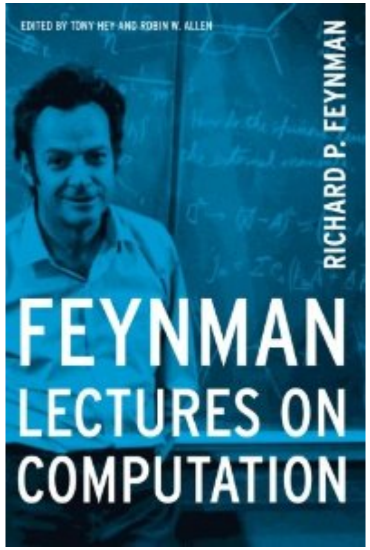
- Numerics: why we care
- Number representation
- Number types for DFEs
- Rounding, Rounding, Rounding,…
- Arithmetic styles
- Error and other numerical issues

Euclids Elements, Representing $a^2+b^2=c^2$

without modern representations, the below is very hard:

Richard Feynman on Computation

In theory, a computer system
can be constructed which uses no energy.



Energy is only needed when **information** is lost.

Reordering of **information** does not require energy from a pure physics perspective.

Of course, moving **information** takes Energy…

Representation of information determines energy consumption for computation!

Why we care about Numerics

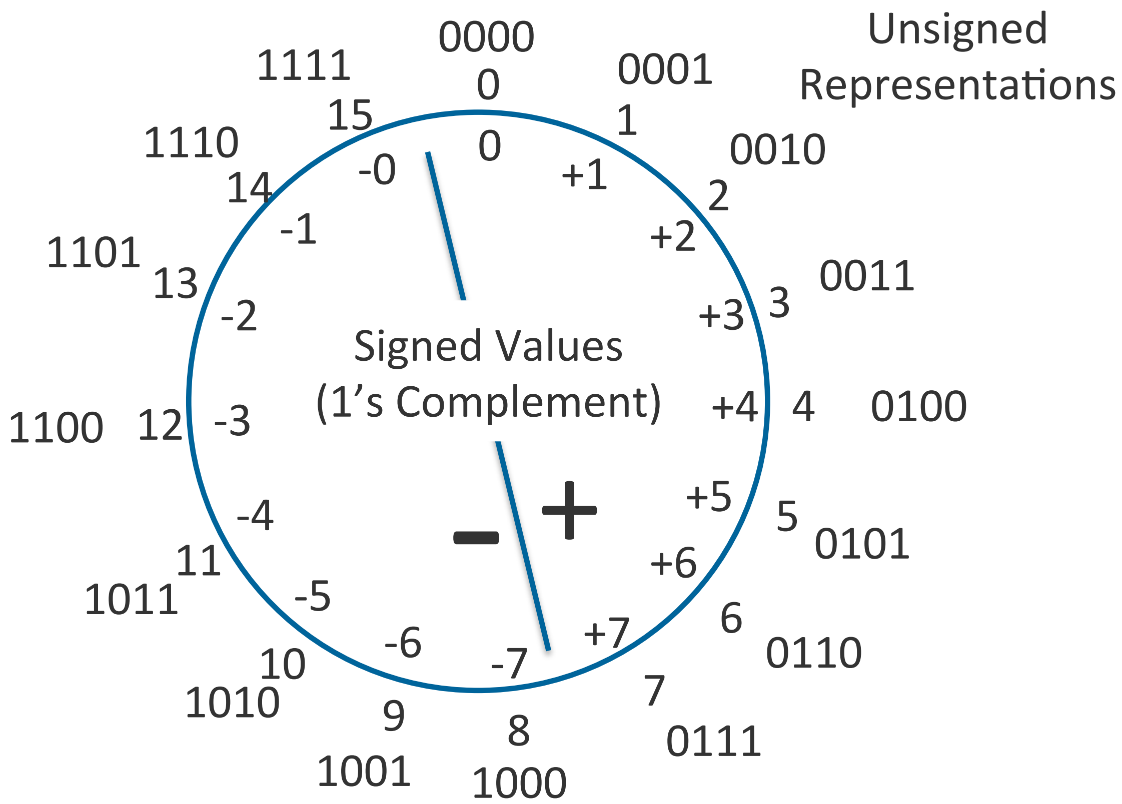
- Performance depends on the number of arithmetic units that fit in space on the DFE
  → lower precision → more units → higher performance
- Accuracy and performance may be competing objectives
  - DFE space depends on data representation and bitwidths
- DFEs give us control over number representation, to achieve just-enough accuracy in minimal space

Number Representation

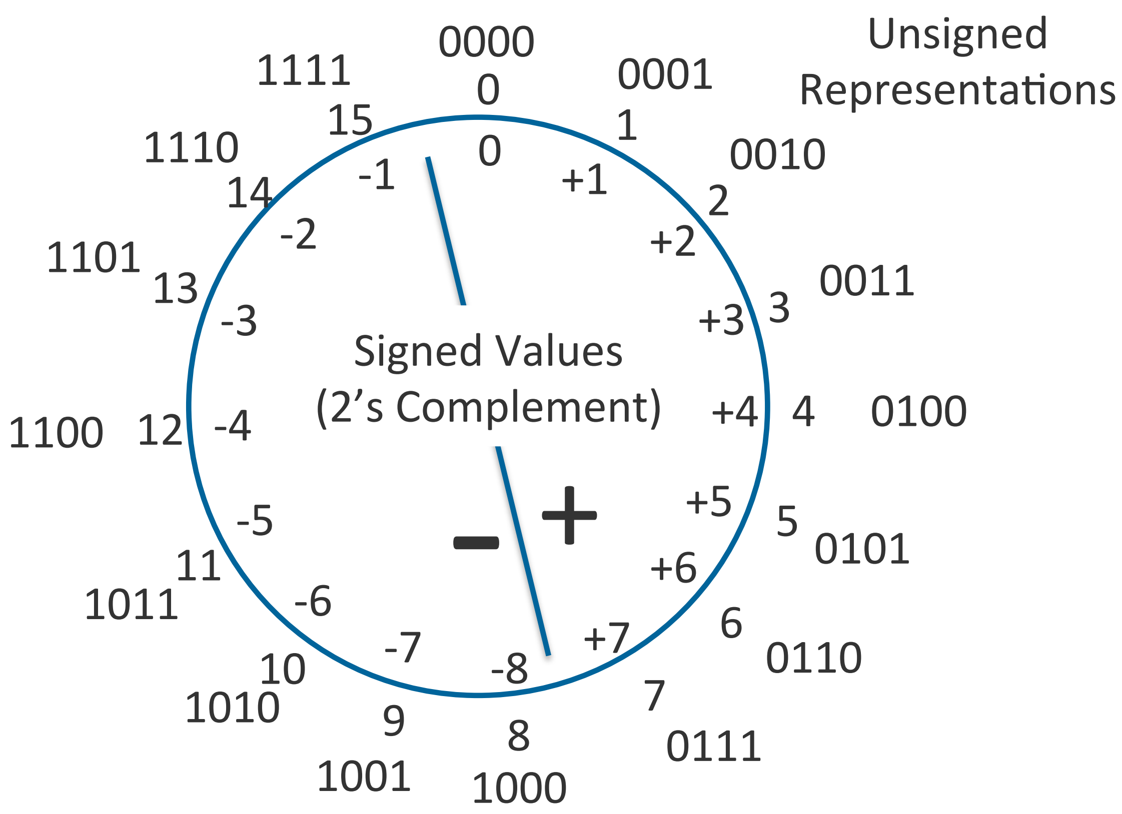
- Microprocessors:
  - Integer: unsigned, one's complement, two's complement,
  - Floating Point: IEEE single-precision, double-precision

- Others:
  - Fixed point
  - Logarithmic number representation
  - Redundant number systems: use more bits, compute faster
    - Signed-digit representation
    - Residue number system (modulo arithmetic)
  - Decimal: decimal floating point, binary coded decimal

One’s Complement



Two’s Complement



Signed N-bit Integers

- Sign-magnitude representation for integer x.

  Most significant bit (msb) is the sign bit.

  Advantages: symmetry around 0, easy access to |x|, simple overflow detection

  Disadvantages: complexity of add/sub.

- One's complement numbers:

  Represent -x by inverting each bit.
  Overflow=Sign_Carry_In ^ Sign_Carry_Cout

  Advantages: fast negation

  Disadvantages: add/sub correction: carry-out of sign bit is added to lsb

- Two's complement:

  Represent -x by inverting each bit and adding '1'.
  Overflow=same as One's c.

  Advantages: fast add/sub

  Disadvantages: magnitude computation requires full addition

Fixed Point Numbers

- Generalisation of integers, with a 'radix point'
- Digits to the right of the radix point represent negative powers of 2

Digit weights: (unsigned)	2^4	2^3	2^2	2^1	2^0 ●	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
	I bits					*F* bits				

- *F* = number of fractional bits
  - Bits to the right of the 'radix point'
  - For integers, F = 0

Fixed Point Mathematics

- Think of each number as: $(V \times 2^{-F})$
- Addition and subtraction: $(V1 \times 2^{-F1}) + (V2 \times 2^{-F2})$
  - Align radix points and compute the same as for integers

	1	0	0	1	0 ●	1	0	0	1	0	
+		1	0	1	1 ●	0	1	0	0	1	0
=	1	1	1	0	1 ●	1	1	0	1	1	0

- Multiplication: $(V1 \times 2^{-F1}) \times (V2 \times 2^{-F2}) = V1{\times}V2 \times 2^{-(F1+F2)}$

					1	0 ●	1	0			
×			1	0	1	0 ●	0	1	0		
=	0	1	1	0	0	1 ●	1	0	1	0	0

- Division: $(V1 \times 2^{-F1}) / (V2 \times 2^{-F2}) = (V1/V2) \times 2^{-(F1-F2)}$

Floating Point Representation

$$\text{sign} \cdot |\text{mantissa}| \cdot base^{\text{exponent}}$$

- regular mantissa = 1.xxxxxx
- denormal numbers get as close to zero as possible: mantissa = 0.xxxxxx with min exponent
- IEEE FP Standard: base=2, single, double, extended widths
- Computing in Space: choose widths of fields + choose base
- Tradeoff:
  - **Performance**: small widths, larger base, truncation.
  - versus **Accuracy**: wide, base=2, round to even.
- Disadvantage: Floating Point arithmetic units tend to be very large compared to Integer/Fixed Point units.

Floating Point Maths

- Addition and subtraction:
  - Align exponents:
    shift smaller mantissa to the larger number's exponent
  - Add mantissas
  - Normalize:
    shift mantissa until starts with '1', adjust exponent
- Multiplication:
  - Multiply mantissas, add exponents
- Division:
  - Divide mantissas, subtract exponents

Number Representation for DFEs

- MaxCompiler has in-built support for floating point and fixed point/integer arithmetic
  - Depends on the *type* of the DFEVar
- Can type inputs, outputs and constants
- Or can *cast* DFEVars from one type to another
- Types are Java objects, just like DFEVars,

```
// Create an input of type t
DFEVar io.input(String name, DFEType t);

// Create an DFEVar of type t with constant value
DFEVar constant.var(DFEType t, double value);

// Cast DFEVar y to type t
DFEVar x = y.cast(DFEType t);
```

DFE Floating Point - dfeFloat

- Floating point numbers with base 2, flexible exponent and mantissa
- Compatible with IEEE floating point **except** does not support denormal numbers
  – When Computing in Space you can use a larger exponent

```
DFEType t = dfeFloat(int exponent_bits, int mantissa_bits);
```

Including the sign bit

- Examples:

	Exponent bits	Mantissa bits
IEEE single precision	8	24
IEEE double precision	11	53
DFE optimized low precision	7	17

Why dfeFloat(7,17)…?

DFE Fixed Point – dfeFixOffset

- Fixed point numbers
- Flexible integer and fraction bits
- Flexible sign mode
  - SignMode.UNSIGNED or SignMode.TWOSCOMPLEMENT

```
DFEType t = dfeFixOffset(int num_bits, int offset, SignMode sm);
```

- Common cases have useful aliases

	Integer bits	Fraction bits	Sign mode
dfeInt(N)	N	0	TWOSCOMPLEMENT
dfeUInt(N)	N	0	UNSIGNED
dfeBool()	1	0	UNSIGNED

Mixed Types

- Can mix different types in a MaxCompiler kernel to use the most appropriate type for each operation
  - Type conversions costs area – must cast manually
- Types can be parameter to a kernel program
  - Can generate the same kernel with different types

```
class MyKernel extends Kernel {
  public MyKernel(KernelParameters k, DFEType t_in, DFEType t_out)
{
     super(k);

     DFEVar p = io.input("p", dfeFloat(8,24));
     DFEVar q = io.input("q", t_in);

     DFEVar r = p * p;

     DFEVar s = r + q.cast(r.getType());
     io.output("s", s.cast(t_out), t_out);
  }
}
```

Rounding

- When we remove bits from the RHS of a number we may want to perform *rounding*.
  - Casting / type conversion
  - Inside arithmetic operations
- Different possibilities
  - TRUNCATE: throw away unwanted bits
  - TONEAR: if >=0.5, round up (add 1)
  - TONEAREVEN: if >0.5 round up, if <0.5 round down, if =0.5 then round to the nearest even number
- Lots of less common alternatives:
  - Towards zero, towards positive infinity, towards negative infinity, random….
- Very important in iterative calculations – may affect convergence behaviour

Rounding in MaxCompiler

- Floating point arithmetic uses TONEAREVEN
- Fixed point rounding is flexible, controlled by the *RoundingMode*
  - TRUNCATE, TONEAR and TONEAREVEN are in-built

```
DFEVar z;
...
optimization.pushRoundingMode(RoundingMode.TRUNCATE);

z = z.cast(smaller_type);

optimization.popRoundingMode();
```

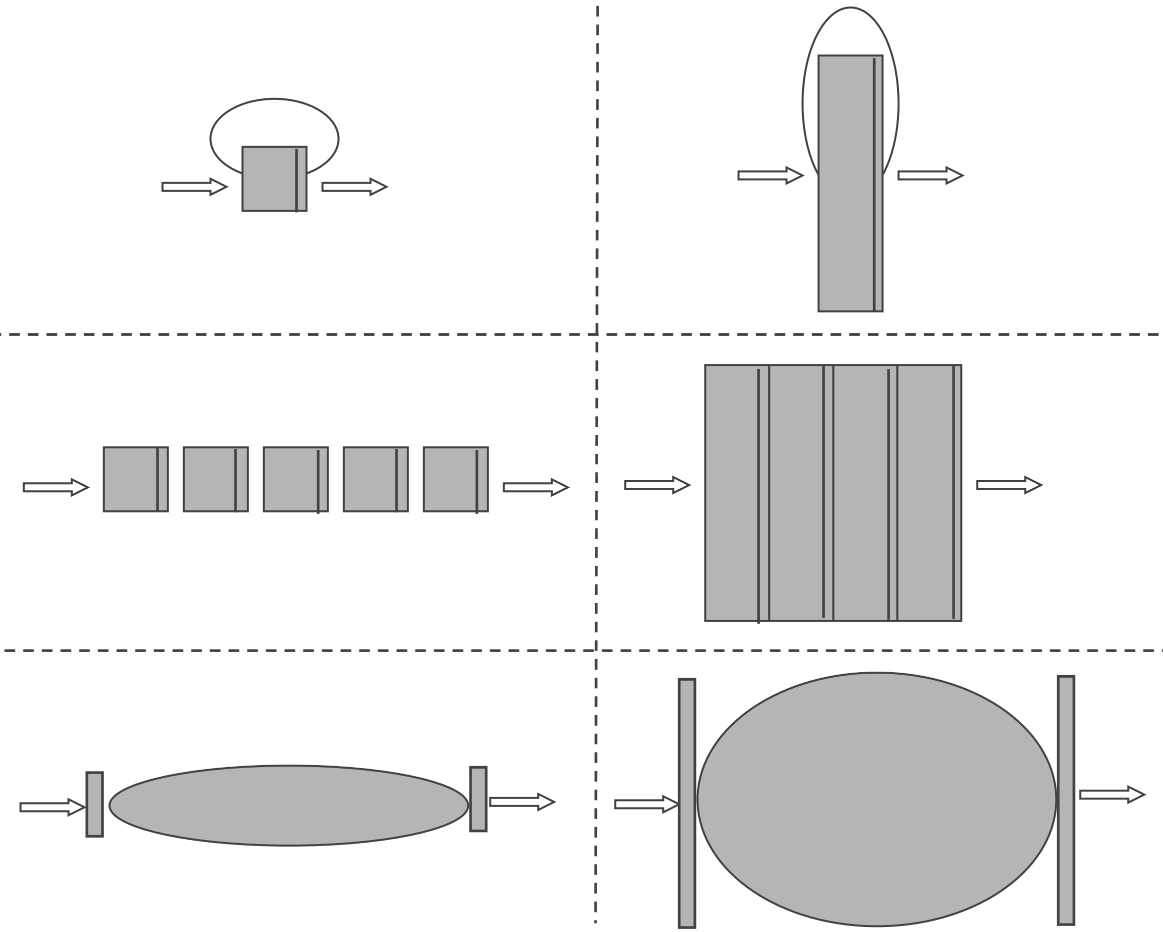
Arithmetic Styles

	Digit-Serial	Digit-Parallel

Sequential
- loop x times

Pipelined
- loop unrolled

Combinational
- loop unrolled
- no registers
- logic min



Arithmetic in MaxCompiler

- By default uses deeply pipelined arithmetic functions
  - Objective is high operating frequency
- Can reduce pipelining gradually to produce combinatorial functions, controlled by *pushing* and *popping* a "pipelining factor"
  - 1.0 = maximum pipelining ; 0.0 = no pipelining

```
DFEVar x, y, z; // floating point numbers
...
z = x * y; // fully pipelined
optimization.pushPipeliningFactor(0.5);
z += x; // half pipelined - lower latency
optimization.pushPipeliningFactor(0.0);
z += y; // no pipelining
optimization.popPipeliningFactor();
optimization.popPipeliningFactor();
z = z * 2; // fully pipelined again
```

Arithmetic takes Space on the DFE

- Addition/subtraction:
  - ~1 logic cell/bit for fixed point,
    while it takes hundreds of logic cells per floating point op
- Multiplication: Can use MULT blocks
  - 18x25bit multiply on Maxeler Vectis DFEs
  - Number of MULTs depends on total bits (fixed point) or mantissa bitwidth (floating point)

Approximate space cost models

	Floating point: dfeFloat(E, M)		**Fixed point: dfeFix(I, F, *TWOSCMP*)**	
	MULTs	**LUTs**	**MULTs**	**LUTs**
Add/subtract	0	$O(\ M\times\log_2(E)\ )$	0	I+F
Multiply	$O(\ \mathrm{ceil}(M/18)^2\ )$	O(E)	$O(\ \mathrm{ceil}((I+F)/18)^2\ )$	0
Divide	0	$O(\ M^2\ )$	0	$O(\ (I+F)^2\ )$

I = Integer bits, F = Fraction bits. E = Exponent bits, M = Mantissa Bits

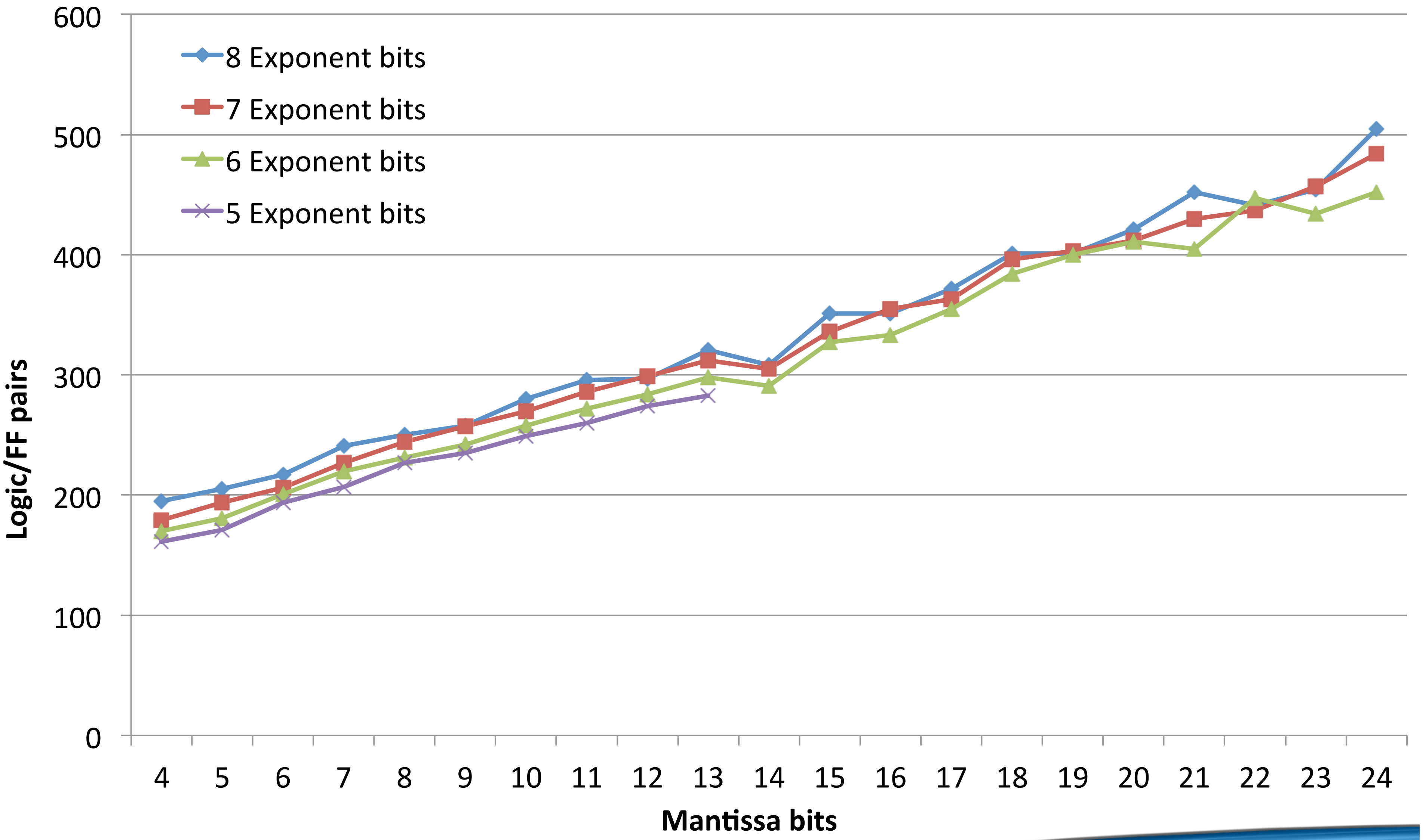
MULT usage for N x M multiplication

M →

N ↓

Bits	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52	54
18	1	1	1	1	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3
20	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3
22	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3
24	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3
26	2	2	2	2	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6
28	2	2	2	2	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6
30	2	2	2	2	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6
32	2	2	2	2	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6
34	2	2	2	2	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6
36	2	3	3	3	4	4	4	4	4	5	5	5	5	6	6	6	6	6	7
38	2	3	3	3	4	4	4	4	4	5	5	5	5	6	6	6	6	6	7
40	2	3	3	3	4	4	4	4	4	5	5	5	5	6	6	6	6	6	7
42	2	3	3	3	4	4	4	4	4	5	5	5	5	6	6	6	6	6	7
44	3	3	3	3	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9
46	3	3	3	3	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9
48	3	3	3	3	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9
50	3	3	3	3	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9
52	3	3	3	3	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9
54	3	4	4	4	6	6	6	6	6	7	7	7	7	9	9	9	9	9	10

Logic usage for floating point addition



Example: Quad Precision Floating Point

Quad Precision (112 bit mantissa, 15 bit exponent)
One Maxeler Maia DFE = 21.4 GFLOP/s
1U MPC-X2000 with 8 Maias = 171.2 GFLOP/s.

1U server node of Sandybridge 16-core is estimated to run at 1.05 GFLOP/s.

1U to 1U: ~160x.

Benefits of Fixed Point

- Consider fixed point
  compared to single precision floating point
- If range is tightly confined,
  we could use *24-bit* fixed point
- If data has a wider range, may need *32-bit* fixed point

	dfeFloat(8,24)	dfeFixOffset(24,..)	dfeFixOffset(32,..)
Add	500 logic cells	24 logic cells	32 logic cells
Multiply	2 MULTs	2 MULTs	4 MULTs

- Arithmetic is not 100% of the chip. In practice, often see ~5x performance boost from fixed point.

Error

- $\forall$A,B: $\Re$. A (op) B = result + error
- Floating point *introduces (dynamic) relative error*
  - Error = f(exponent of result) → relative error
- Fixed point *introduces (static) absolute error*
  - Error = f(rightmost bit position of result) → static error
- Error is minimized by thoughtful rounding

Other numerics issues

- Overflow
  - Number is too large (positive or negative) to be represented
  - Usually catastrophic – important data is lost/invalid
- Underflow
  - Number is too small to be represented and rounds to zero
  - With fixed point, happens gradually
  - With floating point without denormals, happens suddenly
  - Usually underflowing data is not so important (numbers are very small)
- Bias
  - If errors do not have a mean of zero, they will grow with repeated computation.
  - Big issue in iterative calculations
    → numbers gradually get more and more wrong!
  - TONEAREVEN rounding mode minimizes bias

Further Reading on Computer Arithmetic

- Recommended reading:
  - Goldberg, “What Every Computer Scientist Should Know About Floating-Point Arithmetic”, ACM Computing Surveys, March 1991

- Textbooks:
  - Koren, “Computer Arithmetic Algorithms,” 1998.
  - Pahrami, ”Computer Arithmetic: Algorithms and Hardware Designs,” Oxford University Press, 2000.
  - Waser, Flynn, “Introduction for Arithmetic for Digital Systems Designers,” Holt, Rinehard & Winston, 1982.
  - Omondi, “Computer Arithmetic Systems,” Prentice Hall, 1994.
  - Hwang, “Computer Arithmetic: Principles, Architectures and Design,” Wiley, 1978.

Conclusions

- Understand your data requirements
- Know what fixed or floating point you need
- Mind rounding
- Number representations affect the space of computation as well as time!