Imperial College London

100 years of living science

Bridging Some Gaps for FPGA compilation:
Memory Subsystems for Parallelism & Real Number Representations

George A. Constantinides

(joint work with Qiang Liu, Antonio Roldao Lopes, and David Boland)

Circuits and Systems Group

EEE, Imperial College London

http://cas.ee.ic.ac.uk

Outline

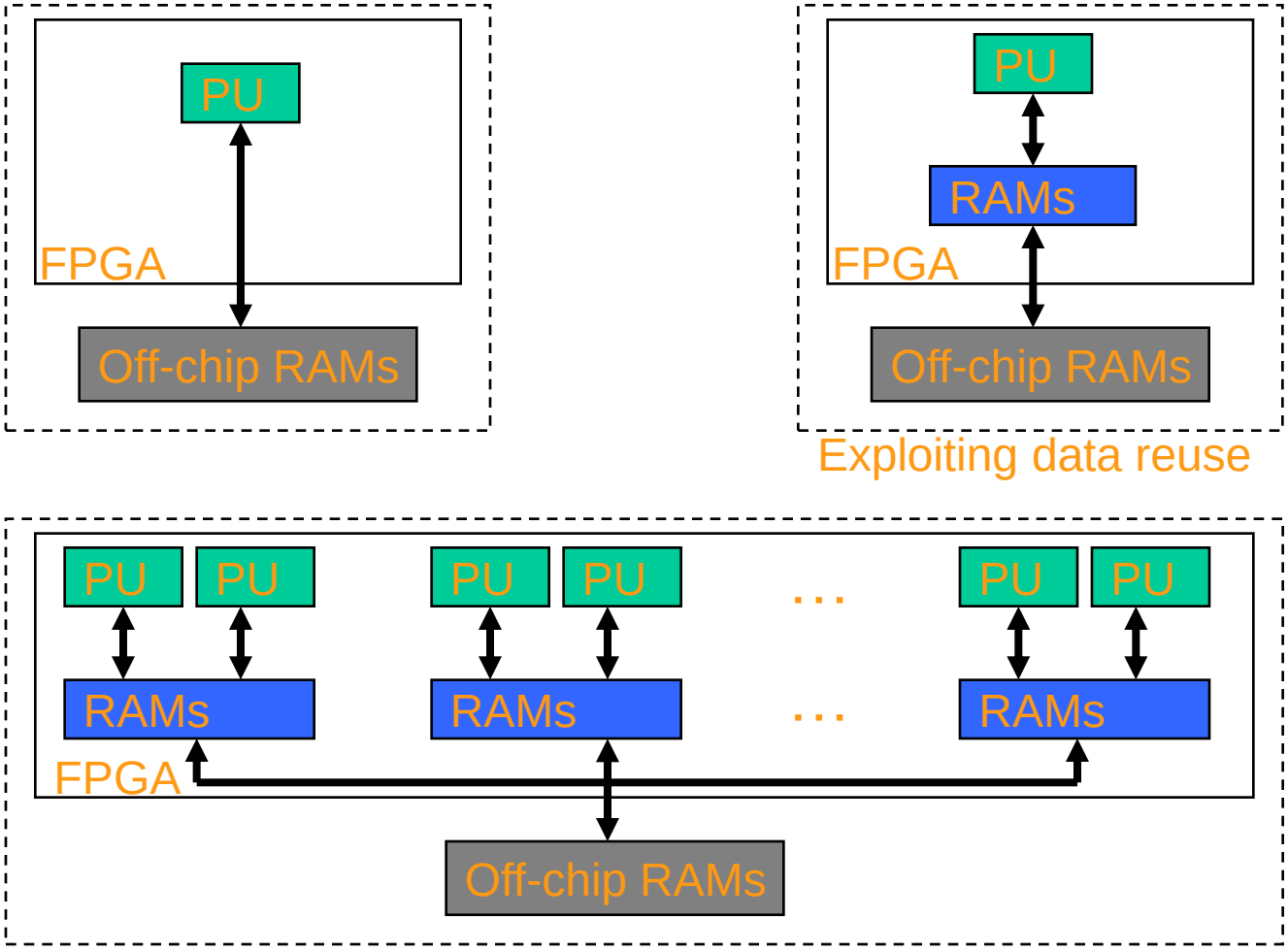
- Combining loop parallelization with scratchpad memory design
  - Customized datpath => customized memory subsystem
  - Formulation as a geometric programming problem
- Real number representations for parallel processing
  - Arbitrary precision algorithm in, finite precision implementation out.
  - Tension between performance and precision
  - Datapath parallelism + specialization => further scope for optimization
  - State-of-the-art in the DSP world
  - Extending: nonlinear arithmetic, iterative methods

Scratchpad and Parallelism: Motivations

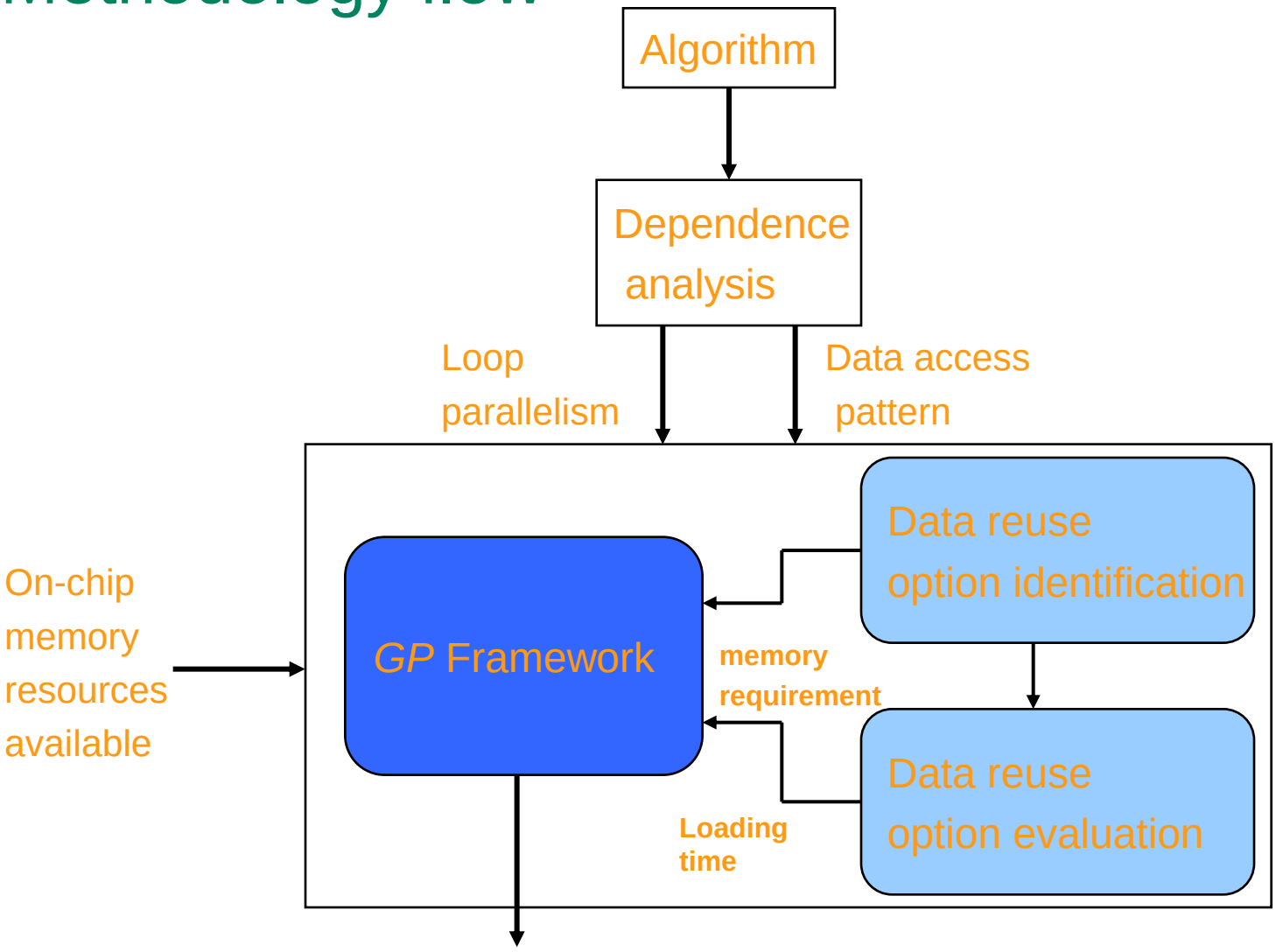
- FPGA-based reconfigurable systems have been applied to a range of applications:
  - DSP, video/audio processing, HPC.

- How to efficiently exploit FPGA resources to achieve an optimal design?
  - Our focus is on memory resources, rather than computational resources.

Motivations—Target platform



Methodology flow



Problem statement

- Imperative code exhibiting data reuse and potential loop-level parallelism
- Data reuse decision: at which levels of a loop nest to insert on-chip arrays to buffer reused data of each array reference
- Parallelization decision: an appropriate strip-mining for each loop, allowing a parameterized degree of parallelism

```
Do I1 =0, U1-1
  Do I2 =0, U2-1
    :
      Do IN =0, UN-1
        ...=A[ f(I1, I2, …, IN) ];
      Enddo;
    :
  Enddo;
Enddo;
```

Off-chip RAM

(a) Original loop structure

```
Do I1 =0, U1-1
  Load( RLA, A );
  Doall I21 =1, kI2
    Do I22 =ceil(U2/kI2)(I21-1),
            min(U2-1, ceil(U2/kI2)I21-1)
      :
        Do IN =0, UN-1
          ...=RLAI21[ g(I22, …, IN) ];
        Enddo;
      :
    Enddo;
  Enddoall;
Endo;
```

On-chip RAM

(b) Loop structure with data reuse arrays *RLA* and strip-mined loop I_2.

Problem statement (Cont.)

- Dependence between data reuse decision and data-level parallelization decision
  - Buffering data on-chip makes parallelization possible

- In which order to make the two decisions?
  - Parallelization decision prior to data reuse decision => infeasibly large on-chip memory requirements
  - Data reuse decision prior to parallelization decision => limited parallelism

Data reuse and loop parallelization decisions should be made at the same time

Problem formulation

$$\min : S\prod_{l=1}^{N}\left\lceil \frac{L_l}{k_l} \right\rceil + \sum_{i=1}^{R}\sum_{j=1}^{E_i}\rho_{ij}C_{ij} \quad (1)$$

subject to

$$\left\lceil \frac{1}{2}\prod_{l=1}^{N}k_l \right\rceil B_{temp} + \left\lceil \frac{1}{2}\prod_{l=1}^{N}k_l \right\rceil \sum_{i=1}^{R}\sum_{j=1}^{E_i}\rho_{ij}B_{ij} \le B \quad (2)$$

$$\sum_{j=1}^{E_i}\rho_{ij} \le 1, \qquad 1 \le i \le R \quad (3)$$

$$\rho_{ij} \in \{0,1\}, \qquad 1 \le j \le E_i,\ 1 \le i \le R \quad (4)$$

$$k_l \ge 1, \qquad 1 \le l \le N \quad (5)$$

$$k_l - (L_l - 1)\sum_{j=1}^{l}\rho_{ij} \le 1,\ 1 \le l \le N, 1 \le j \le E_i,\ 1 \le i \le R \quad (6)$$

ρ_{ij}: data reuse variables

k_l: #partitions of loop *l*

S: #execution cycles of the inner-most loop body

L_l: #iterations of loop *l*

B_{temp}: #on-chip RAM blocks for storing temporary variables (DSA)

B: #on-chip RAM blocks available

(ρ_{ij} and k_l are integer variables and capitals are input parameters)

c.f. Geometric Program

$$\min: \ f_0(x)$$

$$\text{subject to} \ \ f_i(x) \le 1, \ \ i = 1, \ldots, m$$

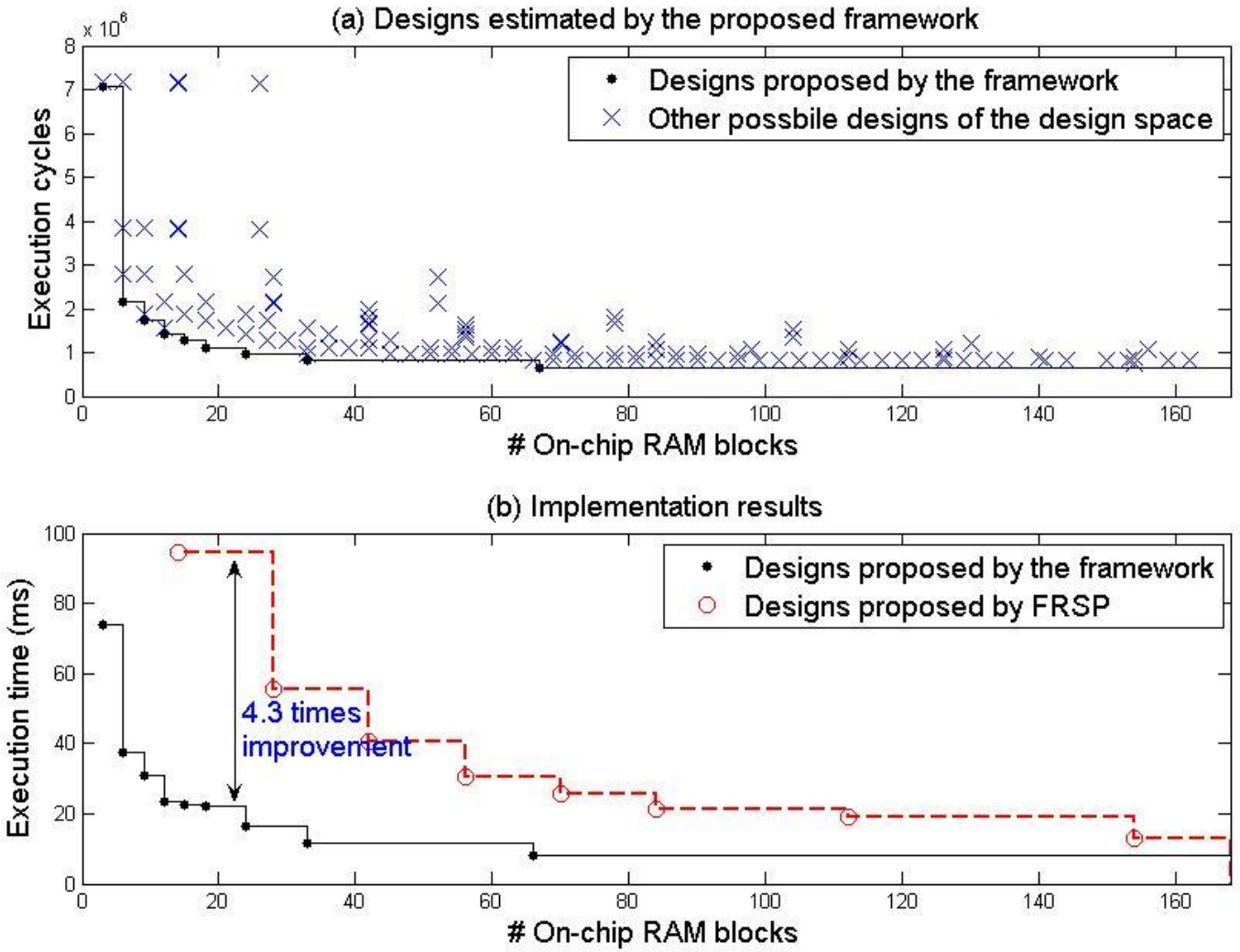
$$h_i(x) = 1, \ \ i = 1, \ldots, p$$

$$\text{where } h(x) = c x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n},$$

$$f(x) = \sum_{k=1}^{K} c_k x_1^{a_{1k}} x_2^{a_{2k}} \cdots x_n^{a_n k}, \ \text{and}$$

$$x \in \Re^n, \ \ x \succ 0, \ \ c, c_k > 0, \ \ a_i, a_{ik} \in R$$

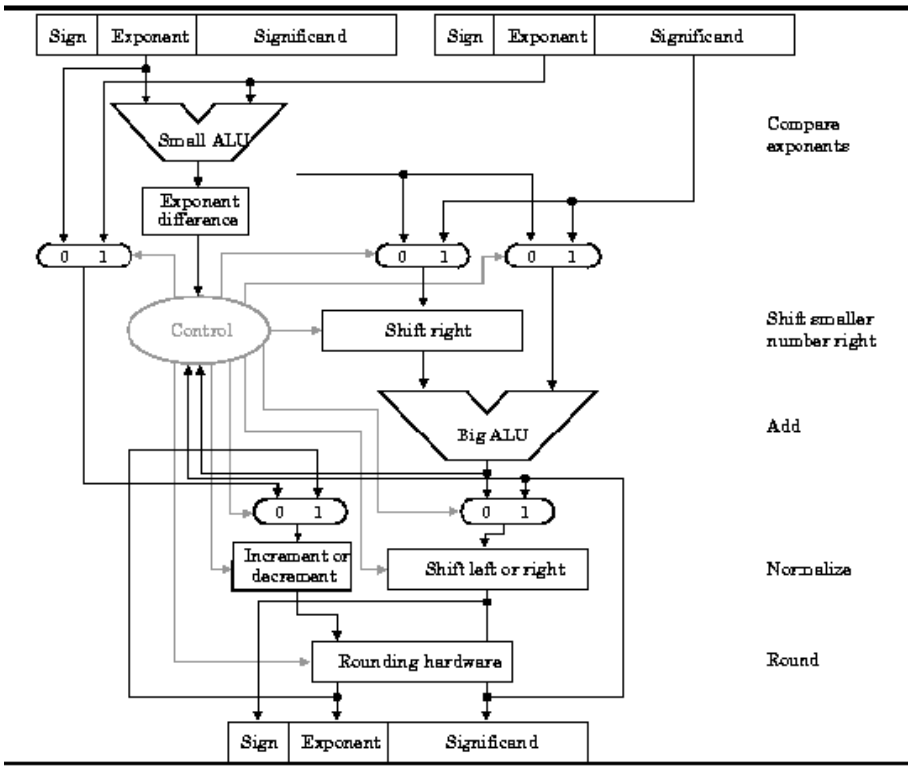
Experimental results—FSME



Finite Precision

- Hardware represents numerical data as bit strings.
- A bit string of length n can represent at most 2^n distinct values.
- Representations vary in how these strings are mapped onto reals: $f : \{0, 1\}^n \rightarrow \mathbb{R}$.
  - Floating-point (s-m-e),
  - Fixed-point (s-m,2c,1c),
  - LNS (s-e),
  - RNS (m,m,...), *etc.*
- We have always cared about using 'enough' precision. Now we should care about using 'just enough' precision.
  - Lower precision $\Rightarrow$ smaller area units, less bandwidth $\Rightarrow$ more units, more transfer $\Rightarrow$ faster.

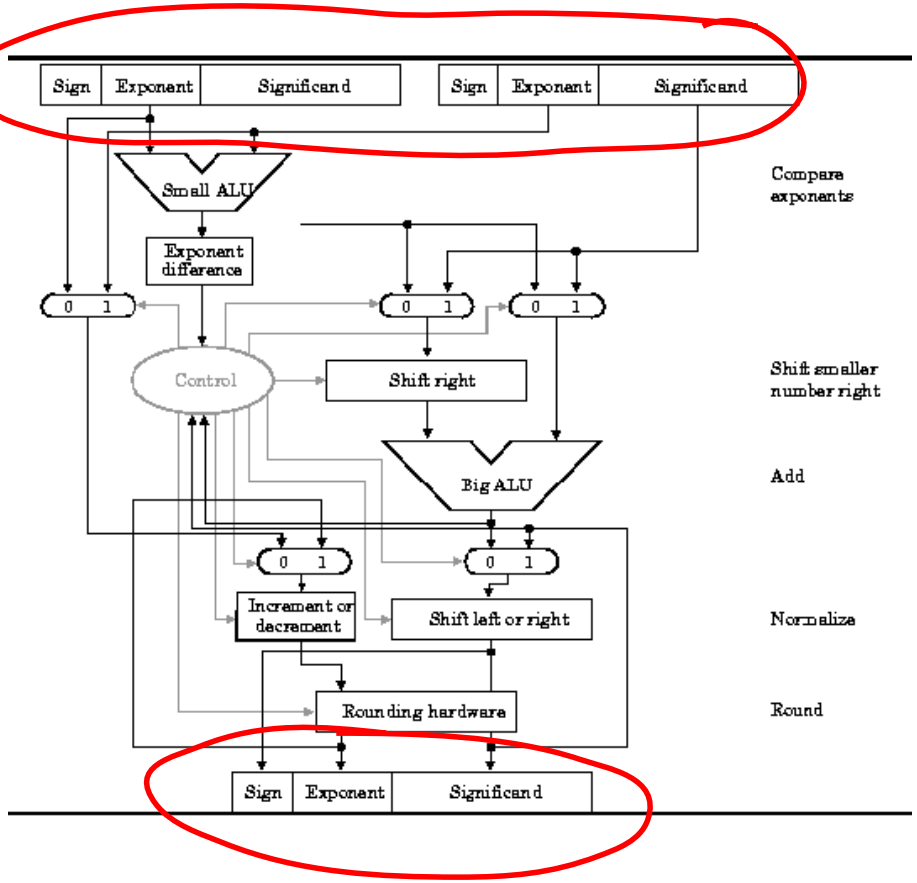
Fixed- and Floating-point Circuit Architectures



(from http://www.cise.ufl.edu/~mssz/CompOrg/CDA-arith.html)

- Floating-point (m,e) bits:
  - $\Theta(m+e)$ bandwidth
  - $\Theta(m^2+e)$ adder (ripple-carry, barrel shift)
  - $\Theta(m^2+e)$ multiplier (array/Wallace/Dadda, ripple-carry)
- Fixed-point n bits:
  - $\Theta(n)$ bandwidth
  - $\Theta(n)$ adder (ripple-carry)
  - $\Theta(n^2)$ multiplier (array/Wallace/Dadda)

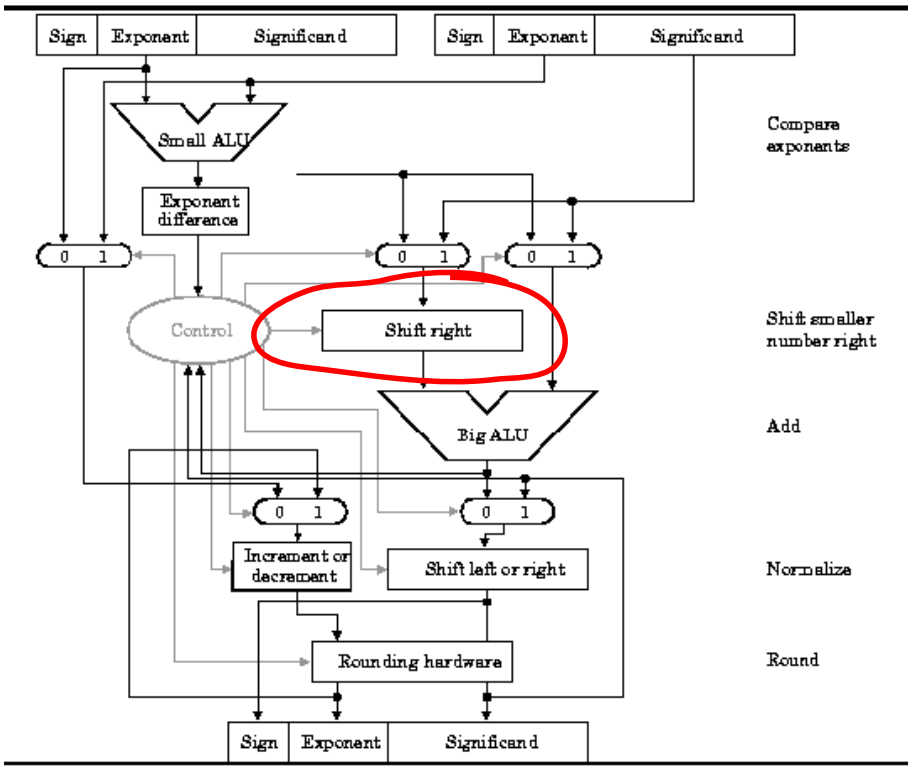
Fixed- and Floating-point Circuit Architectures



(from http://www.cise.ufl.edu/~mssz/CompOrg/CDA-arith.html)

- Floating-point (m,e) bits:
  - $\Theta(m+e)$ bandwidth
  - $\Theta(m^2+e)$ adder (ripple-carry, barrel shift)
  - $\Theta(m^2+e)$ multiplier (array/Wallace/Dadda, ripple-carry)
- Fixed-point n bits:
  - $\Theta(n)$ bandwidth
  - $\Theta(n)$ adder (ripple-carry)
  - $\Theta(n^2)$ multiplier (array/Wallace/Dadda)

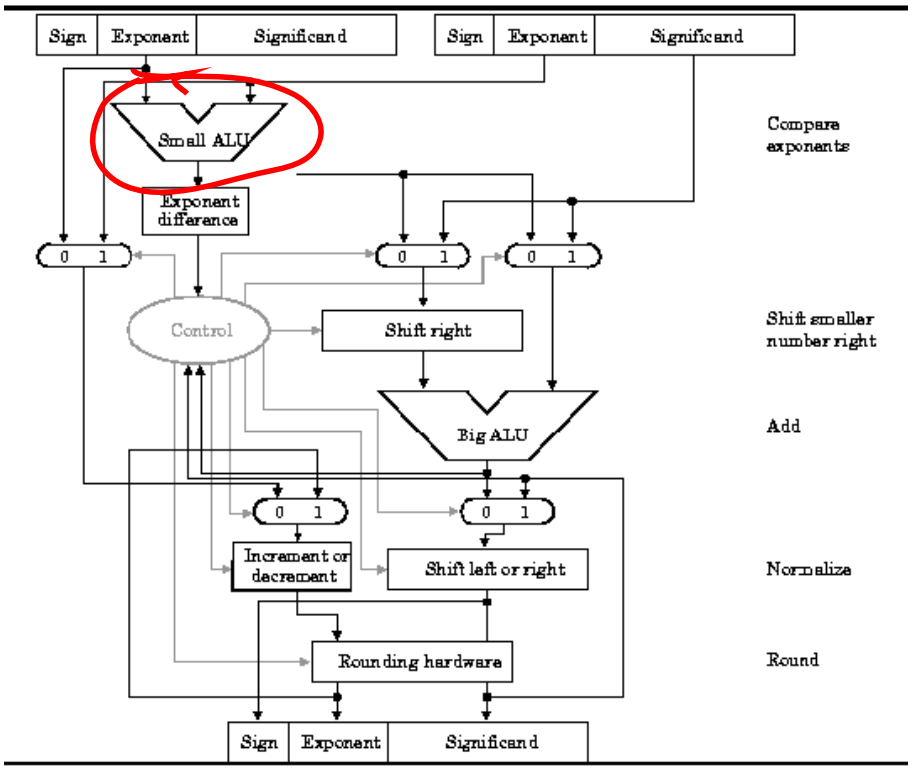
Fixed- and Floating-point Circuit Architectures



(from http://www.cise.ufl.edu/~mssz/CompOrg/CDA-arith.html)

- Floating-point (m,e) bits:
  - $\Theta(m+e)$ bandwidth
  - $\Theta(m^2+e)$ adder (ripple-carry, barrel shift)
  - $\Theta(m^2+e)$ multiplier (array/Wallace/Dadda, ripple-carry)
- Fixed-point n bits:
  - $\Theta(n)$ bandwidth
  - $\Theta(n)$ adder (ripple-carry)
  - $\Theta(n^2)$ multiplier (array/Wallace/Dadda)

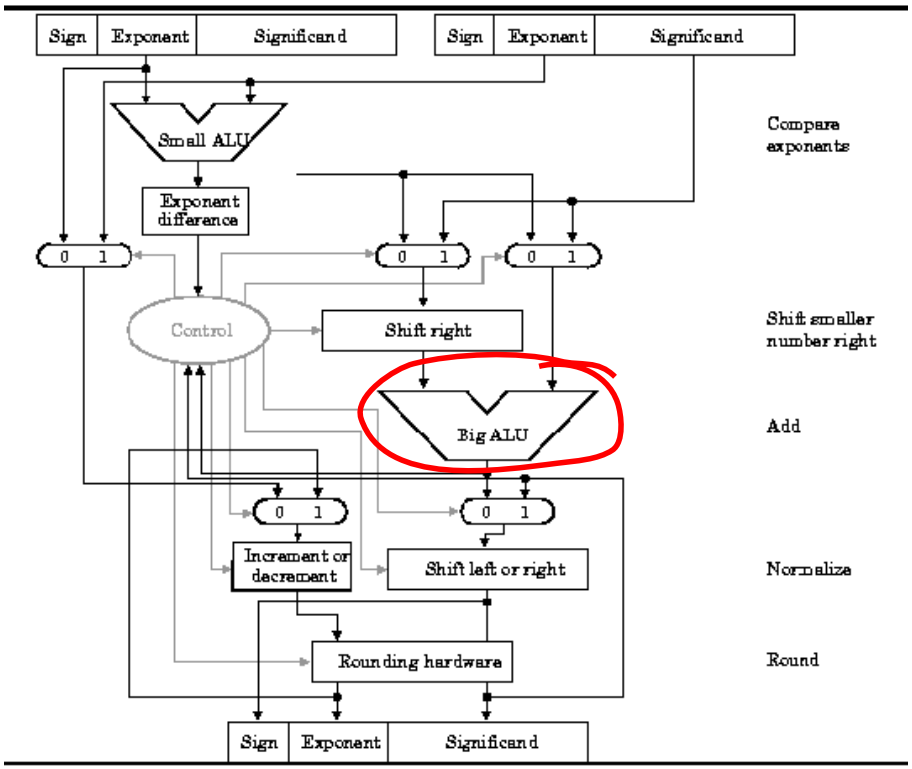
Fixed- and Floating-point Circuit Architectures



(from http://www.cise.ufl.edu/~mssz/CompOrg/CDA-arith.html)

- Floating-point (m,e) bits:
  - $\Theta(m+e)$ bandwidth
  - $\Theta(m^2+e)$ adder (ripple-carry, barrel shift)
  - $\Theta(m^2+e)$ multiplier (array/Wallace/Dadda, ripple-carry)
- Fixed-point n bits:
  - $\Theta(n)$ bandwidth
  - $\Theta(n)$ adder (ripple-carry)
  - $\Theta(n^2)$ multiplier (array/Wallace/Dadda)

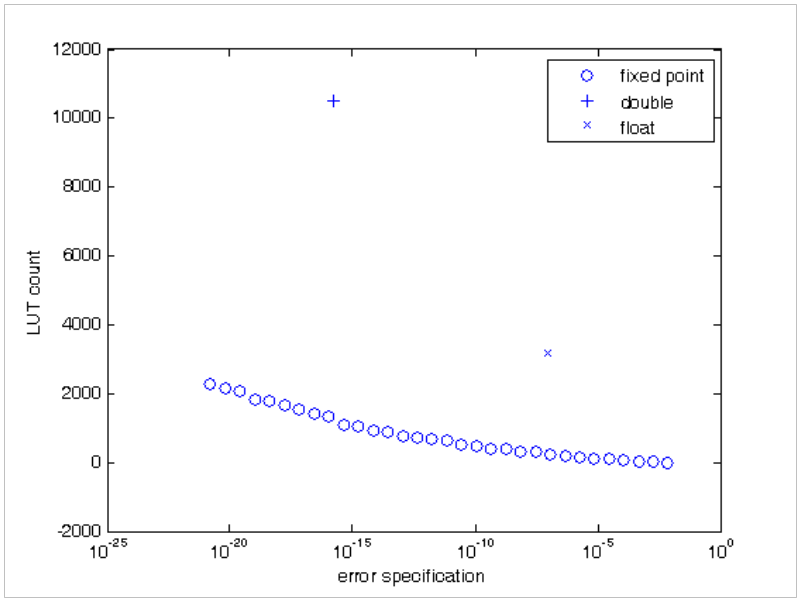
Fixed- and Floating-point Circuit Architectures



(from http://www.cise.ufl.edu/~mssz/CompOrg/CDA-arith.html)

- Floating-point (m,e) bits:
  - $\Theta(m+e)$ bandwidth
  - $\Theta(m^2+e)$ adder (ripple-carry, barrel shift)
  - $\Theta(m^2+e)$ multiplier (array/Wallace/Dadda, ripple-carry)
- Fixed-point n bits:
  - $\Theta(n)$ bandwidth
  - $\Theta(n)$ adder (ripple-carry)
  - $\Theta(n^2)$ multiplier (array/Wallace/Dadda)

Motivation for FPGAs + Fixed-Point: (i) Optimize to Application

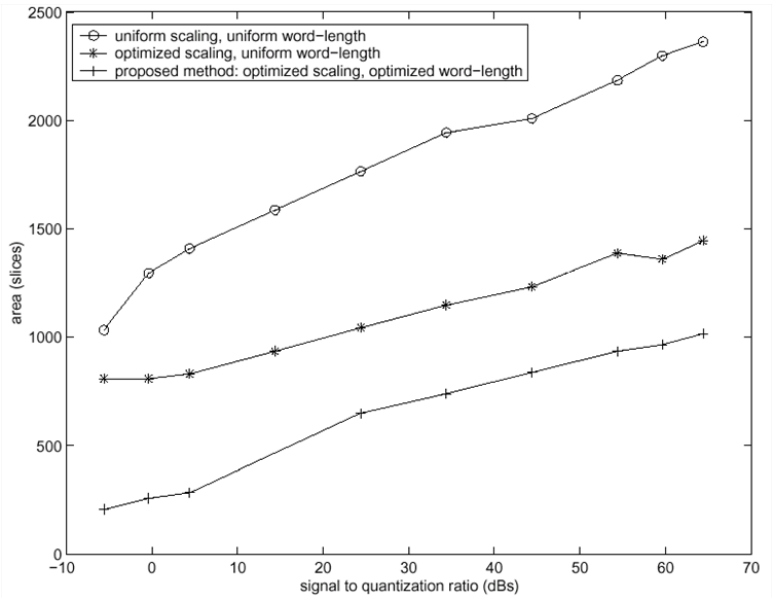


- 10× faster at CG and MINRES[a]
- 33× faster at multivariate GRNG[b]
- 50× faster at shortest path[c]
- 80× faster at learning Bayesian networks[d]
- 100× faster at genome sequencing[e]

---

[a] [Lopes *et al.*, Boland *et al.*]
[b] [Thomas *et al.*]
[c] [Baker *et al.*]
[d] [Pournara *et al.*]
[e] [Storaassli *et al.*]

Motivation for FPGAs + Fixed-Point: (ii) Multiple Word-Length



- Specialisation + Parallelism.
- Provide precision only when the algorithm or data require it.
- An optimization problem:

$$\max_{\mathbf{n}} \text{Perf}(\mathbf{n}) \text{ s.t. } \text{Err}(\mathbf{n}) \leq b \quad (1)$$

Example Code

```
const COEF_MA = 2.125;
const COEF_AR = 0.1875;

double x, y;

y = 0.0;

while(1) {
   read(x);
   y = COEF_MA*x + COEF_AR*y;
   write(y);
}
```

The Same Code in Multiple Word-length Fixed-Point

```
const COEF_MA = ((signed 6)17);
const COEF_AR = ((signed 3)3);

signed 10 x;
signed 10 t1;
signed 12 t2;
signed 14 y;

y = 0;

while(1) {
   read(x);
   t1 = (x*COEF_MA)(14 downto 5);        // LSB-quant
   t2 = y(13 downto 4)*COEF_AR;          // ...and here
   y = ((t1@zeros(3 downto 0)) + t2)(13 downto 0);
                                         // align + MSB-adj
   write(y);
}
```

The Same Code in Multiple Word-length Fixed-Point

```
const COEF_MA = ((signed 6)17);
const COEF_AR = ((signed 3)3);

signed 10 x;
signed 10 t1;
signed 12 t2;
signed 14 y;

y = 0;

while(1) {
   read(x);
   t1 = (x*COEF_MA)(14 downto 5);        // LSB-quant
   t2 = y(13 downto 4)*COEF_AR;          // ...and here
   y = ((t1@zeros(3 downto 0)) + t2)(13 downto 0);
                                         // align + MSB-adj
   write(y);
}
```

2.125 * 8 = 17
0.1875 * 16 = 3

The Same Code in Multiple Word-length Fixed-Point

```
const COEF_MA = ((signed 6)17);
const COEF_AR = ((signed 3)3);

signed 10 x;
signed 10 t1;
signed 12 t2;
signed 14 y;

y = 0;

while(1) {
   read(x);
   t1 = (x*COEF_MA)(14 downto 5);        // LSB-quant
   t2 = y(13 downto 4)*COEF_AR;          // ...and here
   y = ((t1@zeros(3 downto 0)) + t2)(13 downto 0);
                                         // align + MSB-adj
   write(y);
}
```

lose 5 LSBs

The Same Code in Multiple Word-length Fixed-Point

```
const COEF_MA = ((signed 6)17);
const COEF_AR = ((signed 3)3);

signed 10 x;
signed 10 t1;
signed 12 t2;
signed 14 y;

y = 0;

while(1) {
   read(x);
   t1 = (x*COEF_MA)(14 downto 5);          // LSB-quant
   t2 = y(13 downto 4)*COEF_AR;            // ...and here
   y = ((t1@zeros(3 downto 0)) + t2)(13 downto 0);
                                           // align + MSB-adj
   write(y);
}
```

manual alignment of binary-point location

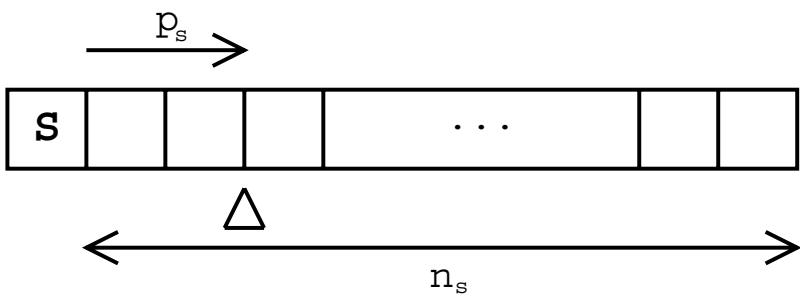
The Same Code in Multiple Word-length Fixed-Point

```
const COEF_MA = ((signed 6)17);
const COEF_AR = ((signed 3)3);

signed 10 x;
signed 10 t1;
signed 12 t2;
signed 14 y;

y = 0;

while(1) {
   read(x);
   t1 = (x*COEF_MA)(14 downto 5);          // LSB-quant
   t2 = y(13 downto 4)*COEF_AR;            // ...and here
   y = ((t1@zeros(3 downto 0)) + t2)(13 downto 0);
                                           // align + MSB-adj
   write(y);
}
```

using range calculation, we know that bit 14 is never used

The Two Problems

- We want to do this automatically! How?
- Range Estimation
  - How do we know the bounds on each variable?
  - We need these to define an appropriate location for the binary point. $p_s = \lfloor \log_2 |P_s| \rfloor + 1$.
  - They can also enable circuit optimization, *e.g.* the previous MSB-adjustment.
- Roundoff Error Estimation
  - To address word-length optimization, we need to define $Err(\mathbf{n})$.
- There are several approaches to each problem: we will look at some.



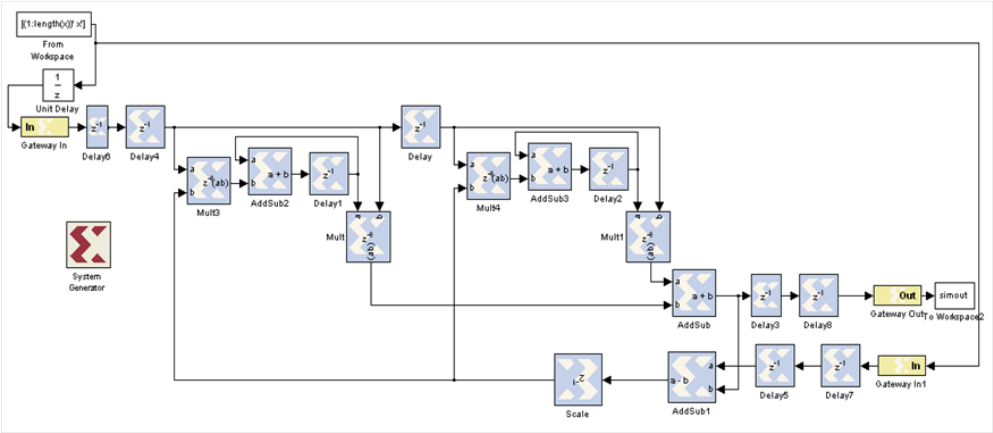
Linear Time-Invariant Computation

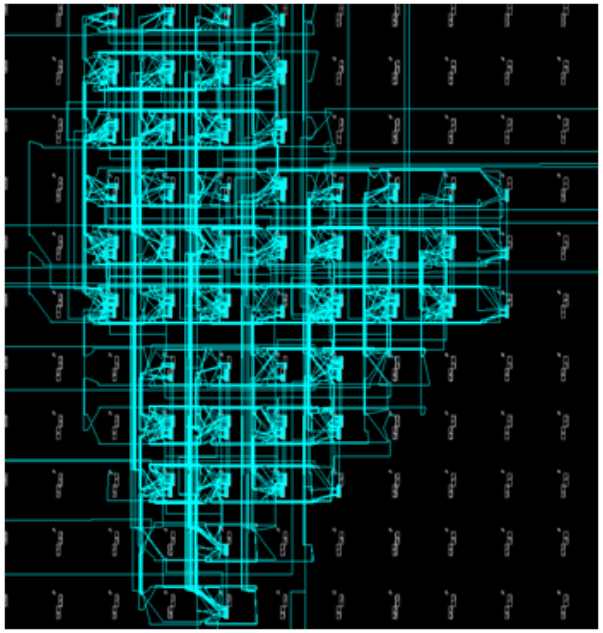
- Historically, DSP has been dominated by LTI computation.
  - LTI computation admits special solution techniques to both problems.
- For simplicity, consider a computation as having one input $x[t]$, which can change over 'time', or 'iteration', and one output $y[t]$.
- The computation C acts to transform x into y, $y = C(x)$.
- The computation is said to be linear iff:
  - $\forall \alpha, x, y.\ y = C(x) \rightarrow \alpha y = C(\alpha x)$ (homogeneity) and
  - $\forall x_1, x_2, y_1, y_2.\ y_1 = C(x_1) \wedge y_2 = C(x_2) \rightarrow y_1 + y_2 = C(x_1 + x_2)$ (additivity).
- Define the time-shift operator $y = D_\tau(x)$ by $y[t] = x[t - \tau]$.
- The computation is said to be time-invariant iff:
  - $\forall x, y, \tau.\ y = C(x) \rightarrow D_\tau(y) = C(D_\tau(x))$.

Range estimation: other options

- If the computation is non-LTI, analytical options are more limited. A common choice is interval arithmetic.
  - Let $x \in [-1, 1]$. $(x-1)^2$?
    Option A: $[-2, 0]^2 = [0, 4]$.
    Option B:
    $[-1, 1]^2 - 2[-1, 1] + 1 = [0, 1] + [-2, 2] + [1, 1] = [-1, 4]$.
- To mitigate the dependence problem, significant recent interest in affine arithmetic.
  - Represent every signal as $y = c_0 + \sum_{i=1}^{k} c_k \epsilon_k$, where $\epsilon_k \in [-1, 1]$ is an *unknown symbolic constant*.
  - Only successful if the computation itself is affine.
- Recent work on SMT.
- We are working on polynomial arithmetic.

Tools


From Workspace
Unit Delay
In
Gateway In
Delay6
Delay4
Mult3
AddSub2
Delay1
Mult
Delay
Mult4
AddSub3
Delay2
Mult1
System Generator
AddSub
Delay3
Delay8
Out
Gateway Out
simout
To Workspace2
Scale
AddSub1
Delay5
Delay7
In
Gateway In1



Conclusion

- There is a big gap in abstraction between industrial standards for hardware design and software design.
- We have focused on two aspects of that gap:
  - Automating the memory subsystem design
  - Automating the number representation design.
- Scope for interaction?
  - Termination proofs in finite precision arithmetic.
  - Incorporating notions of numerical correctness in programming languages.
  - Static analysis for memory subsystem design, targeting non-affine (or non-loop based) code.
  - Always interested in applications, especially from HPC.