Ascenium: A Continuously Reconfigurable Architecture

Robert Mykland
Founder/CTO
robert@ascenium.com
August, 2005
Ascenium: A Continuously Reconfigurable Processor

- Continuously reconfigurable approach provides:
  - The computational efficiency of direct logic implementations in ASICs
  - All the flexibility of microprocessors (pure software)

- Unique architecture enables the array to be effectively targeted by an ANSI Standard C compiler
  - CPU logic is continuously redefined to match the compiler’s needs for each instruction (≈20 cycles)

- Massive parallelism even in programs with little or no inherent instruction-level parallelism
  - Dozens of lines of sequential C code are executed every instruction
Ascenium Block Diagram

- No instruction set
- RLU is combinatorial logic that settles
- Memory is accessed in statically scheduled block operations
- Two instruction streams
  1) Memory
  2) Configuration
- Whole loops are often executed as one RLU instruction
Ascenium Basic Operation

12 RISC Instructions

165

ALU
Register
File
Memory

Conventional

Control

Reconfigurable Logic Unit (RLU)

In
Out
Memory

Ascenium

Config Control

Memory Control

Clock = 24

Conventional

Ascenium
**A128X16 Rough Layout**

- 130 nm process
- 128 16-bit logic elements
- 64 MACs/clock cycle
- 200 MHz DDR II = 3.2 GB/s external memory bandwidth
- 256KB onchip memory
- 19.2 GB/s peak onchip memory bandwidth
- 500 mW @ 200 MHz
- 52 mm$^2$
Logic Element

Input A (16 bits) → \( \frac{1}{2} \) 16-Bit Full Multiplier - Accumulator → Fast Carry → LUT X (16 bits) → Output X (16 bits)

Input B (16 bits) → LUT Y (16 bits) → Output Y (16 bits)

Input C (16 bits) → Wide Logic → Iterator Y (16 bits) → Output Y (16 bits)

Iterator X (16 bits) → LUT X (16 bits) → Output X (16 bits)

Iterator Y (16 bits) → Wide Logic → Iterator Z (16 bits)

Logic Element

ascenium corporation
Array Connectivity
Compiler Block Diagram

GCC Based Compiler (open source) → Library (open source) → Bytecode Linker (open source) → Bytecode

Bytecode files

Proprietary Ascenium Code Generator (develop) → Ascenium Executable

Source code C, C++, Java
Code Generator Diagram

LLVM PARSER → GENERIC INT. REP. (GIR) → GLOBAL THREADING → GIR OPTS.

ASCENIUM INT. REP. (AIR) → FUNC. ASSIGNMENT & SCORING → ARRAY INST. PACKING & COMP. → DATA DIRECTIVES

FETCH ORDER ASSY. → AIR OPTS. → OUTPUT

● indicates where optimizations occur
Generic Intermediate Representation (GIR)
FFT Inner Loop C Code

\[
i_1 = i_0 + n_2; \\
i_2 = i_1 + n_2; \\
i_3 = i_2 + n_2; \\
r_1 = x[2 * i_0] + x[2 * i_2]; \\
r_2 = x[2 * i_0] - x[2 * i_2]; \\
t = x[2 * i_1] + x[2 * i_3]; \\
x[2 * i_0] = r_1 + t; \\
r_1 = r_1 - t; \\
s_1 = x[2 * i_0 + 1] + x[2 * i_2 + 1]; \\
s_2 = x[2 * i_0 + 1] - x[2 * i_2 + 1]; \\
t = x[2 * i_1 + 1] + x[2 * i_3 + 1]; \\
x[2 * i_0 + 1] = s_1 + t; \\
s_1 = s_1 - t; \\
x[2 * i_2] = (r_1 * \text{co2} + s_1 * \text{si2}) >>15; \\
x[2 * i_2 + 1] = (s_1 * \text{co2} - r_1 * \text{si2}) >>15; \\
t = x[2 * i_1 + 1] - x[2 * i_3 + 1]; \\
r_1 = r_2 + t; \\
r_2 = r_2 - t; \\
t = x[2 * i_1] - x[2 * i_3]; \\
s_1 = s_2 - t; \\
s_2 = s_2 + t; \\
x[2 * i_1] = (r_1 * \text{co1} + s_1 * \text{si1}) >>15; \\
x[2 * i_1 + 1] = (s_1 * \text{co1} - r_1 * \text{si1}) >>15; \\
x[2 * i_3] = (r_2 * \text{co3} + s_2 * \text{si3}) >>15; \\
x[2 * i_3 + 1] = (s_2 * \text{co3} - r_2 * \text{si3}) >>15; \\
\]
FFT Loop Array Instruction

```
FFT Loop Array Instruction

A X0 X2
B Y0 Y2
C X1 X3
D Y1 Y3
E C1 S1
F X0 Y0
G X1 Y1
H C2 S2
I C3 S3
J Y3 Y6
K X3 X6
L Y3 Y6
M Y3 Y6
N Y3 Y6
O Y3 Y6
P Y3 Y6
Q Y3 Y6
R Y3 Y6
S Y3 Y6
T Y3 Y6
U Y3 Y6
V Y3 Y6
W Y3 Y6
X Y3 Y6
Y Y3 Y6
Z Y3 Y6
AA Y3 Y6
BB Y3 Y6
CC Y3 Y6
DD Y3 Y6
EE Y3 Y6
FF Y3 Y6
GG Y3 Y6
HH Y3 Y6
II Y3 Y6
JJ Y3 Y6
KK Y3 Y6
LL Y3 Y6
MM Y3 Y6
```
FFT Loop Data Directives

1. Fetch the array instruction and data directives
2. Load the array instruction
3. Read \(x[t]\) in four 64-bit wide banks 32 times
4. Write \(x[t]\) in one 256-bit wide bank 32 times, changing banks every 8 operations
5. Repeat steps 3 and 4 three more times
6. At the same time as 3–5, read in \(w\) coefficients
7. Pass 1: \(w\) every clock; 2: \(w\) every 4 clocks, then every 16, then every 64
8. Stop
### DSP Benchmark Results (Simulated, Hand-Coded)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>A128X16 core, 130nm, 250MHz, 250mW</th>
<th>TMS320C64x core, 130nm, 300MHz*&lt;sup&gt;1&lt;/sup&gt;, 250mW</th>
</tr>
</thead>
<tbody>
<tr>
<td>256-Point Radix-4 FFT</td>
<td>512 ns</td>
<td>8,920 ns</td>
</tr>
<tr>
<td>Real Block FIR Filter</td>
<td>512 ns</td>
<td>6,827 ns</td>
</tr>
<tr>
<td>Complex Block FIR Filter</td>
<td>512 ns</td>
<td>6,827 ns</td>
</tr>
<tr>
<td>DCT x 24</td>
<td>512 ns</td>
<td>6,080 ns</td>
</tr>
<tr>
<td>IIR Filter NR = 2,048</td>
<td>292 ns</td>
<td>6,827 ns</td>
</tr>
</tbody>
</table>

<sup>1</sup> The 130nm C64x core will operate up to 720 MHz, but burns more power at higher frequencies.
Ascenium Status

- Patents filed
- Chip architecture complete, polishing ongoing
- Prototype compiler complete & demo-able
- Development plan in place
- Raising seed round
Summary

• Great performance and performance/mW:
  >10x programmable DSPs
  >4x performance/mW of FPGAs

• Great time-to-market:
  -- Normal GPP software development cycle

• Great ease-of-use:
  -- Programmers writing standard, non-optimized code in C/C++ get all the performance