Catapult C® Synthesis

Creating Parallel Hardware from C++

Andres Takach

Chief Scientist, C-Based Design

February 2008

Mentor Graphics®

Agenda

- Implementing bit-accurate data types in C++
- Implementing parallelism from sequential C++
  - A simple FIR filter
  - Saturation and rounding implications
- Creating pipelined hierarchical systems
  - Discrete Cosine Transform
    - 1 pixel per clock cycle throughput

Bit Accurate Data Types

- Hardware Designers need exact bit widths
  - Extra bits costs gates ($$) and performance ($$)
- C++ data types are insufficient for modeling and have ambiguities
- DSP designers use rounding and saturation
  - Hardware engineers generally do not, unless pressed
- SystemC data types have ambiguities and limitations for algorithm modeling
- Mentor Graphics Algorithmic C data types provide superior vehicle for bit accurate hardware design

Mentor Graphics "Algorithmic C" types

- Fixed-point and Integer types
- Faster simulation
  - Up to 200x faster than SystemC types
- Consistent, with no ambiguities
- Parameterized
  - Facilitate reusable algorithmic development
- Built in Rounding and Saturation modes
- Usable within a SystemC environment
- Freely available for anyone to download

http://www.mentor.com/products/c-based_design

Templatized AC Fixed Data Types

```
ac_fixed<W,I,S,Q,O> my_variable
```

- `W` = Overall Width
- `I` = Number of integer bits
- `S` = signed or unsigned (boolean)
- `Q` = Quantization mode
- `O` = Overflow mode

```
ac_fixed<8,1,true,AC_RND,AC_SAT> my_variable ;
    "0.0000000" 8-bit signed, round & saturate
ac_fixed<8,8,true,AC_TRN,AC_WRAP> my_variable ;
    "00000000" 8-bit signed, no fractional bits.
```

Using C++ For Parallel Hardware

- Function call with all I/O on the interface
  - Represents the I/O of the hardware to be built
- C++ allows object-oriented reusable hardware
  - Technology and Frequency independent
  - Multiple instantiations of functions with state
    - Just like RTL component instantiation
  - Instantiations with different implementations
    - Like VHDL architectures
  - Enables resource sharing across time and function
    - Unlike RTL

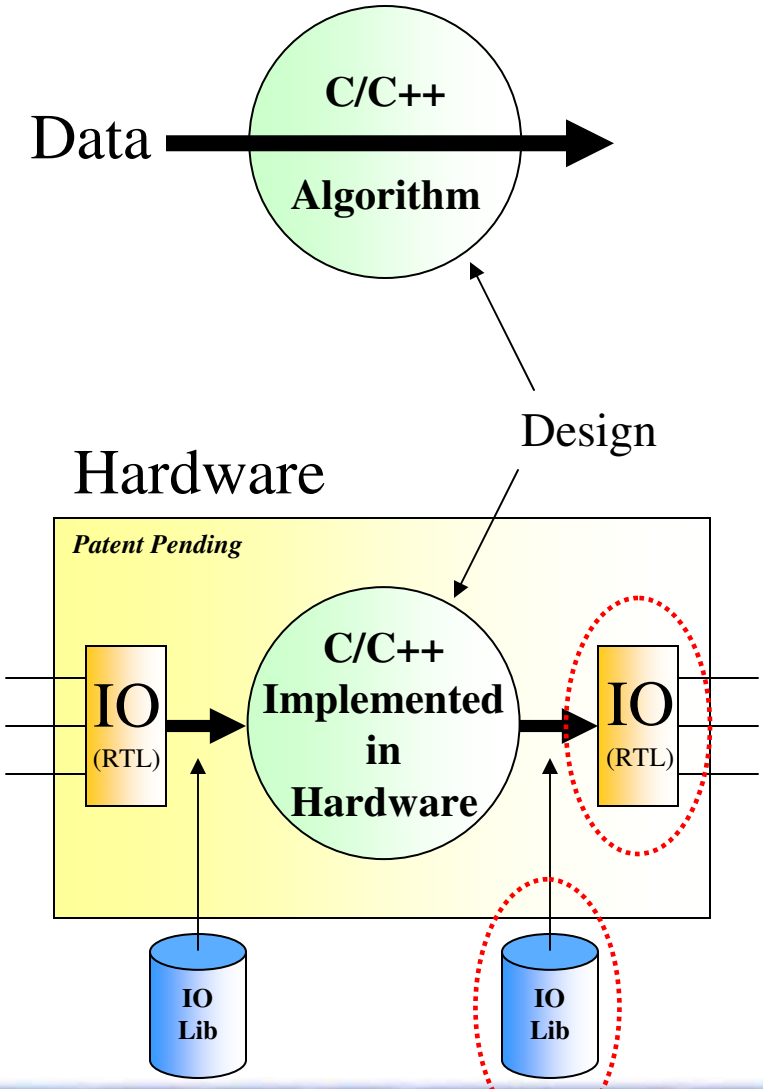
A Simple FIR Filter Model

```
void fir_filter (
     IN_TYPE       &input,
     COEFF_TYPE    coeffs[NUM_TAPS],
     OUT_TYPE      &output
) {

     static IN_TYPE regs[NUM_TAPS];
     MAC_TYPE temp = 0.0;

     SHIFT:for (int i=NUM_TAPS-1 ; i>=0; i--) {
          if (i==0) regs[i] = input ;
          else regs[i] = regs[i-1] ;
     }

     MAC:for (int j = 0 ; j < NUM_TAPS ; j++ )
          temp += regs[j] * coeffs[j] ;

     output = temp ;
}
```

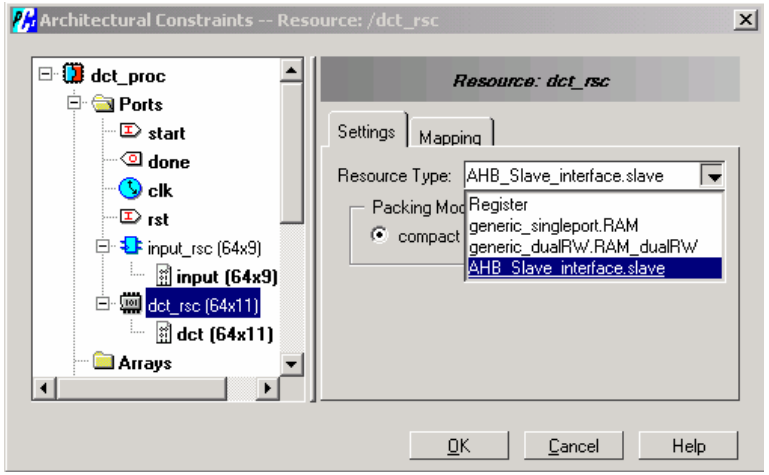
- **Input, coefficients and output**
- **Static taps**
- **MAC type for full precision**
- **Output type for optional rounding and saturation**

Defining The Hardware Interface

Patented Interface synthesis technology makes it possible



- Untimed C++ has no concept of time



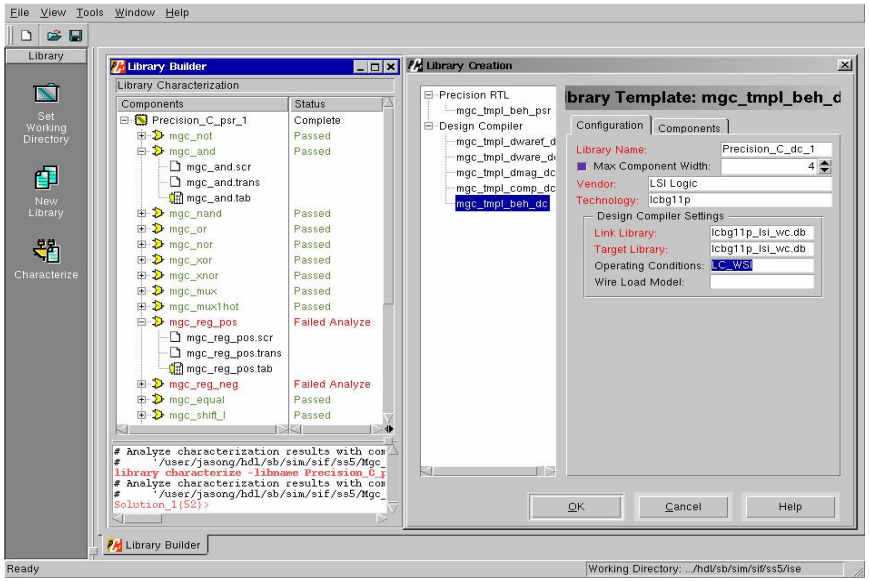
- How does this help?
  - ANY interface is possible
  - Design is built to the interface
  - C++ source remains independent of the interface

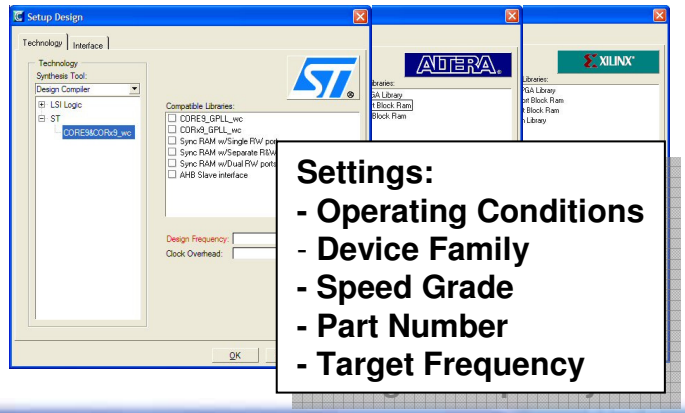
Optimizing Untimed C++

- Catapult maps user-selected physical resources for each variable in the C++ code
  - Wires, handshakes, registers, RAM's, custom interfaces, custom components
- Catapult builds efficient hardware optimized to the constraints of resource bandwidth
- Catapult enables you to quickly find architectural bottlenecks in an algorithm
- Datapath pipelines are created automatically in order to meet desired operating frequency

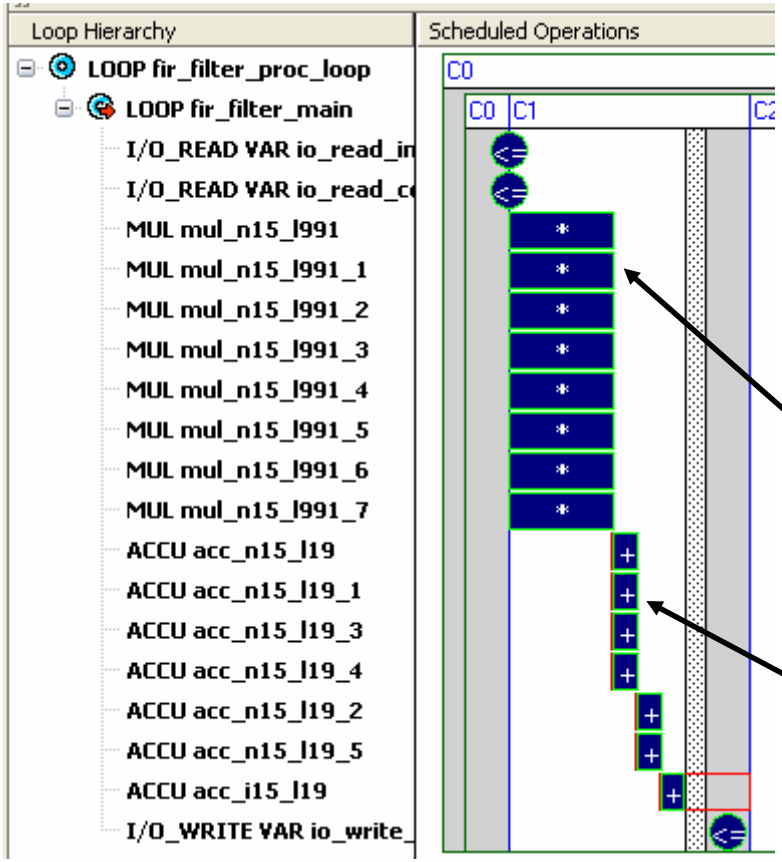
Technology Driven Synthesis

- The exact ASIC or FPGA technology is “characterized” for accurate timing and area metrics for operators of differing bit widths
- Algorithms are scheduled using these technology specific libraries
  - Like having a synthesis timing guru creating your RTL
- Key for high quality, technology-optimized designs with specific operating frequency requirements





FIR Filter Unrolled For Parallelism



- 8 multipliers and an adder tree
- Optimization for latency results in fast components at the chosen operating frequency (250 MHz, 180nm)
- ASIC synthesis typically offers 4 operator area/speed possibilities

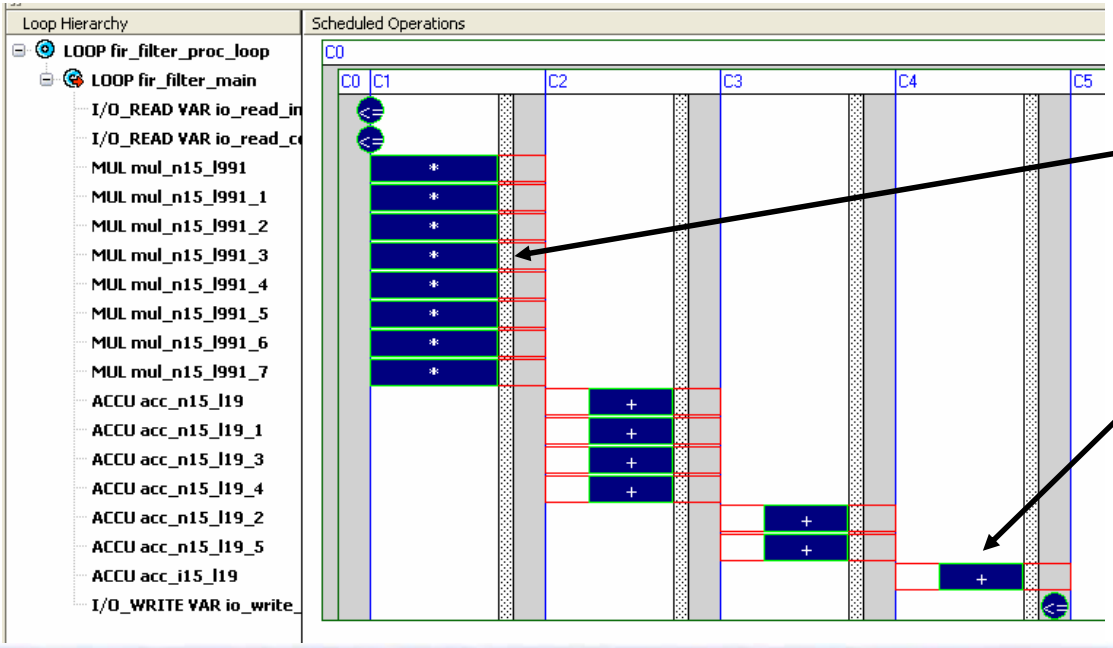
Fastest multiplier in technology

Fastest adders in technology

103,000 sq. microns

Optimizing For Throughput

- Same C++ code can be scheduled to use smaller multipliers
  - Smaller adders too
- Pipelining still keeps throughput data rate
- 40% saving in area after synthesis of completely different RTL
- Characterized target library leverages technology timing



Slower multiplier in technology

Slower adders in technology

63,000 sq. microns

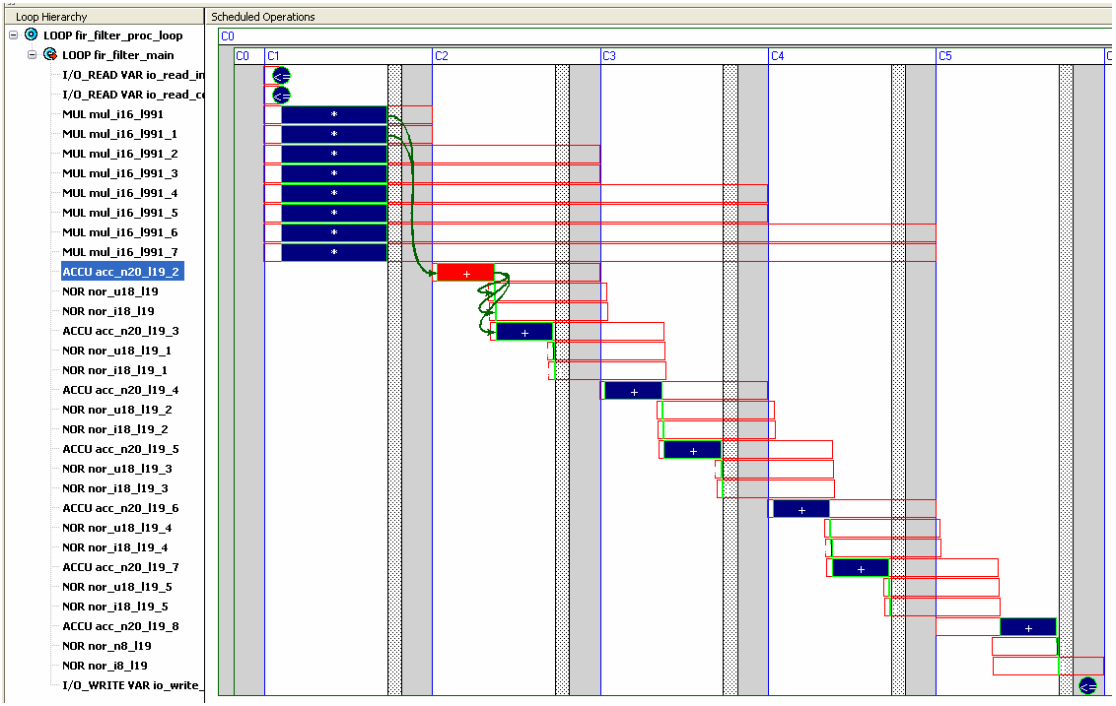
Partial Unrolling

- Allows “n” copies of the loop body to be created in parallel
- 1, 2, 3, 4 copies will give different throughputs
  - 1 => 8 cycles
  - 2 => 4 cycles
  - 3 => 3 cycles (3,3,2)
  - 4 => 2 cycles
- This assumes that all I/O has enough bandwidth
  - With FIR filters, the coefficients (if programmable) and tap storage (registers or RAM’s) are key

The Trouble With Saturation

- Saturation is order-dependent and undesirable when creating parallelism
  - e.g. with an 8-bit signed integer storage (-128 to 127)
    - (100 + 50 -50) is not the same as (100 -50 +50)
  - Creates dependency chains
- Rounding adds carry-in further downstream
- Preferable to do arithmetic at higher precision
  - And then round and truncate at the end

Rounding And Saturating Accumulator



- Creates long string of arithmetic as it must be done in the same sequential order as the C++ to guarantee algorithm match
- Larger area & lower performance than using full precision and rounding at the end

8x8 Discrete Cosine Transform

- **Simple two-dimensional "numerical recipe" implementation**
  - **Rows, then columns, with intermediate storage array**

```
#include "constants.h"
void dct_float(double input[8][8], double dct[8][8]) {

  double temp[8][8];
  double tmp;

  mult1:for (int y=0; y < 8; y++ )
    middle1:for (int x=0; x < 8; x++ ) {
      tmp = 0;
      inner1:for (int i=0; i < 8; i++ )
        tmp = tmp + input[y][i] * coeff[x][i];
      temp[y][x] = tmp;
    }

  mult2:for (int x=0 ; x < 8; x++ )
    middle2:for (int y=0; y < 8; y++ ) {
      tmp = 0;
      inner2:for (int i=0 ; i < 8 ; i++ )
        tmp = tmp + temp[i][x] * coeff[y][i];
      dct[y][x] = tmp/32;
    }
}
```

Multiply-accumulate 512 times each = 1024 Multiplies

One nested loop set for rows

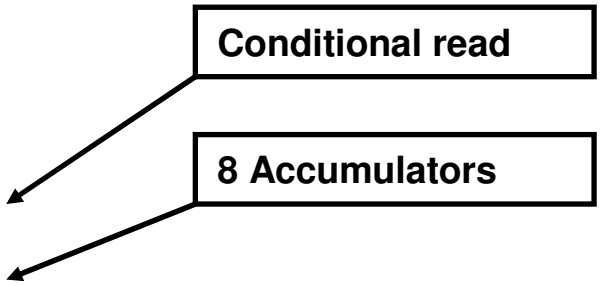
Second nested loop set for columns – outputs data by columns sequentially

Hardware Design with C++

- Algorithmic Synthesis maintains memory architecture
  - Shift register or…
  - Circular buffer
- Just unrolling sequential algorithms may not yield optimum parallel hardware architecture
- C++ code must reflect memory accesses required for desired hardware architecture

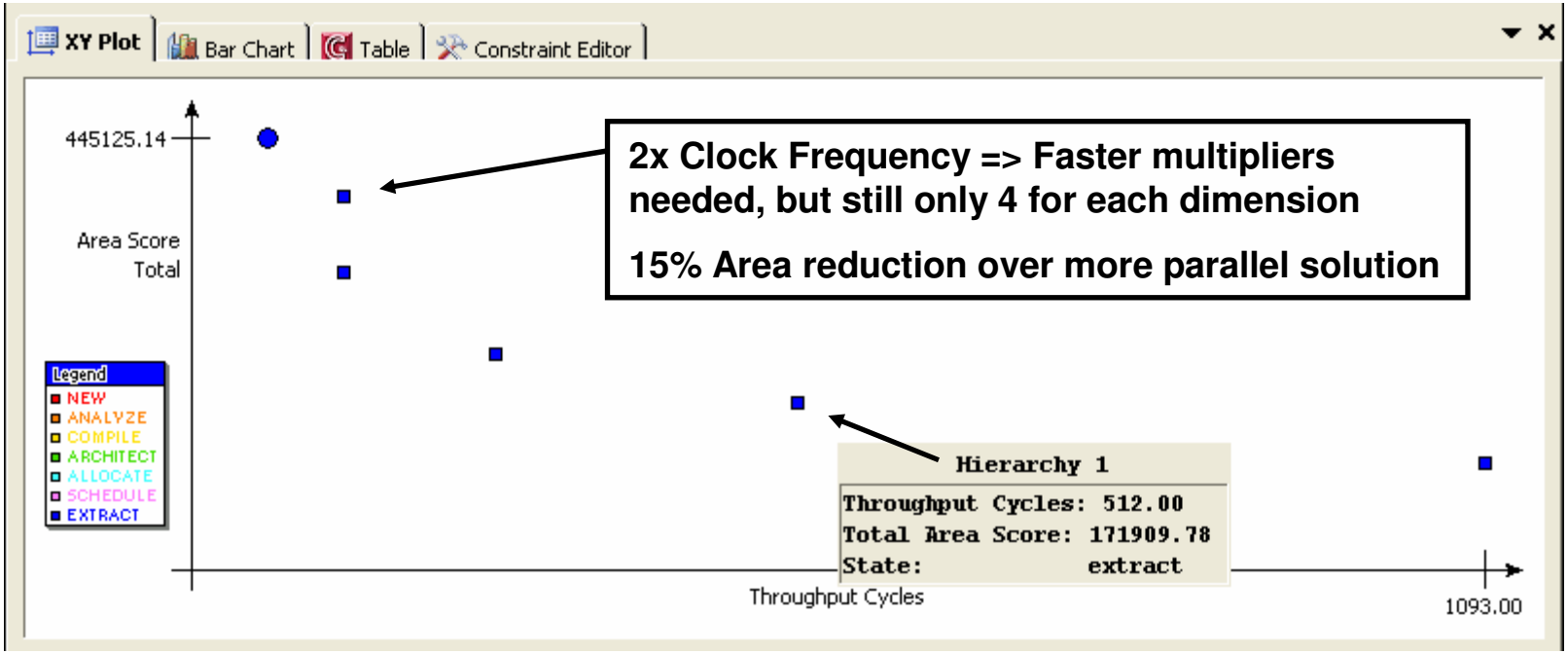
2D DCT for hardware

```
void dct_rows (
     pixel_in_t input[8][8],
     temp_t temp[8][8]
) {
     temp_t tmp[8] ;
     pixel_in_t pixel_read ;
     outer1:for (int y=0; y < 8; y++ ) {
          middle1:for (int x=0; x < 8; x++ ) {
               inner:for (int i=0; i < 8; i++ ) {
                    if (i==0) pixel_read = input[y][x] ;
                    if (x==0) tmp[i] = 0 ;
                    tmp[i] += pixel_read * coeff[i][x] ;
                    if (x==7) temp[y][i] = tmp[i] ;
               }
          }
     }
}
```



- Read inputs in linear order to allow streaming of data
  - Avoid reads of same memory location
- Parallel accumulators change the algorithm architecture

X/Y plot for 180nm at 90MHz



XY Plot | Bar Chart | **Table** | Constraint Editor

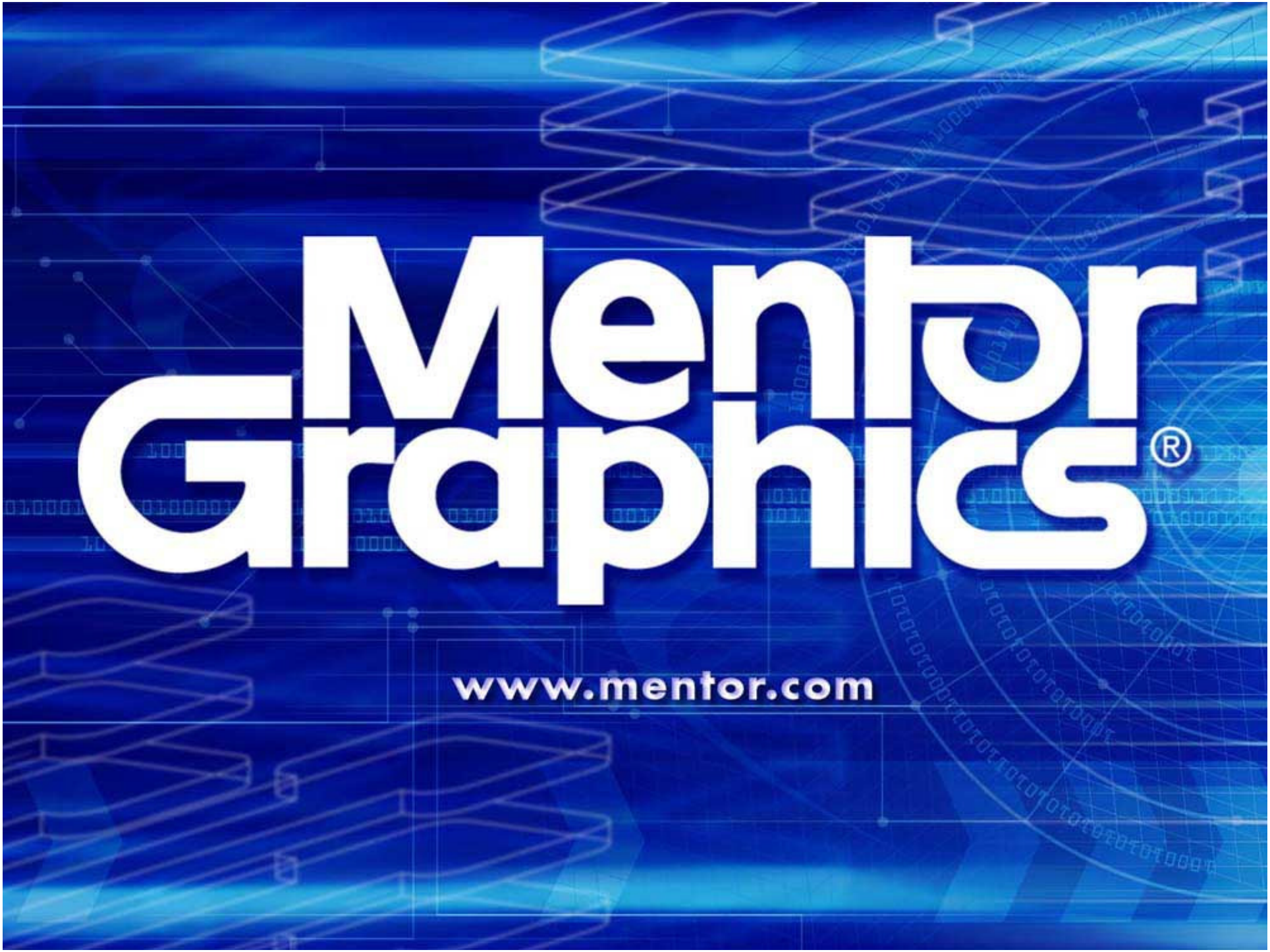
Solution	Status	Total Area Score	Latency Cycles	Latency Time	Throughput Cycles	Throughput Time
One Multiplier	Passed extract	110221.65	1093	12144.44	1093	12144.44
Hierarchy 1	Passed extract	171909.78	1534	17044.44	512	5688.89
Hierarchy 2	Passed extract	222619.16	770	8555.56	256	2844.44
Hierarchy 4	Passed extract	386888.59	388	4311.11	128	1422.22
Hierarchy 8	Passed extract	445125.14	197	2188.89	64	711.11
Hierarchy 4 2x Clock	Passed extract	384655.19	390	2166.67	128	711.11

FPGA Targets

- Catapult's core usage centers around ASIC design
- FPGA's require unique optimization and mapping to achieve high performance results
- FPGA multipliers are fixed in performance capability
  - 9x9 or 18x18 may not fit bit widths exactly
    - 10x8 65nm pipelined multiplier: ~250 pS
    - Virtex5 DSP48E or Stratix III DSP ~2000 pS
- Greater parallelism at lower frequencies than ASIC target may be desirable
- Technology-aware pipelining can alleviate throughput at the expense of latency
- Fabric arithmetic (carry chains) often limit Fmax
- Catapult's new FPGA accelerated libraries produce results which approach the theoretical max for FPGA frequency

Summary

- Highly Parallel bit-accurate hardware implementations are being implemented with commercial ESL tools today
- Pure ANSI C++ is familiar and requires no proprietary tools for development
- Technology-aware High-Level-Synthesis allows algorithm retargeting at an abstraction much higher than RTL
- Implementation is based on target technology characteristics yielding more efficient hardware
  - More parallelism vs higher clock rate


Mentor Graphics®
www.mentor.com