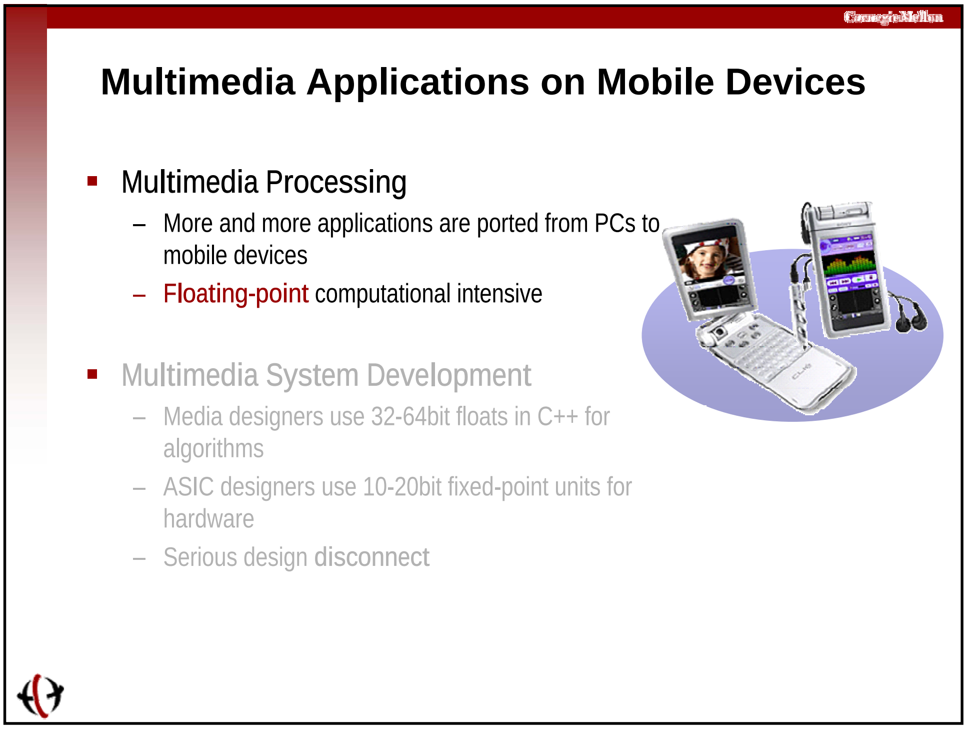
Lightweight Arithmetic for Mobile Multimedia Devices

Tsuhan Chen 陳祖翰
Carnegie Mellon University
tsuhan@cmu.edu

Thanks to Fang Fang and Rob Rutenbar



Multimedia Applications on Mobile Devices

- Multimedia Processing
  - More and more applications are ported from PCs to mobile devices
  - Floating-point computational intensive
- Multimedia System Development
  - Media designers use 32-64bit floats in C++ for algorithms
  - ASIC designers use 10-20bit fixed-point units for hardware
  - Serious design disconnect

Multimedia Applications on Mobile Devices

- Multimedia Processing
  - More and more applications are ported from PCs to mobile devices
  - Floating-point computational intensive

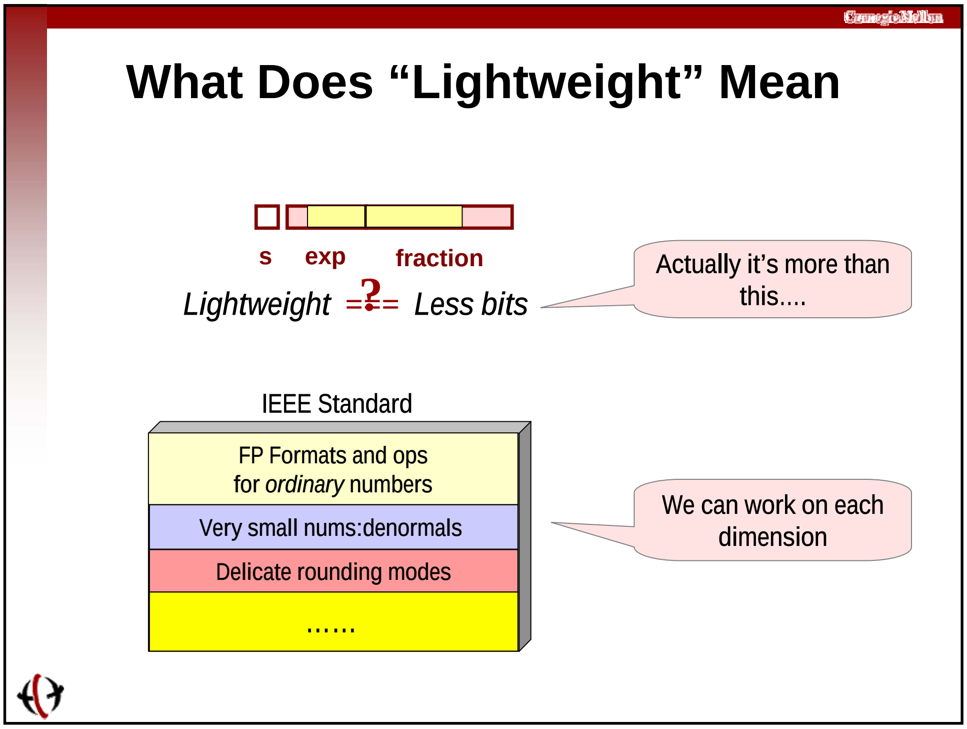
- Multimedia System Development
  - Media designers use 32-64bit floats in C++ for algorithms
  - ASIC designers use 10-20bit fixed-point units in hardware
  - Serious design disconnect

Fixed-Point vs. Floating-Point

Fixed point	Floating-point
s Integer fraction	s exp fraction
× Limited dynamic range and precision	✓ Wide dynamic range & high precision
✓ Small, less power consumption	× Big, power intensive
× From SW to HW: time-consuming and error-prone	✓ Easy translation from SW to HW

How about make this lightweight?

Don’t use more than necessary .



What Does “Lightweight” Mean

s exp fraction

Lightweight $\overset{?}{==}$ *Less bits*

Actually it’s more than this....

IEEE Standard

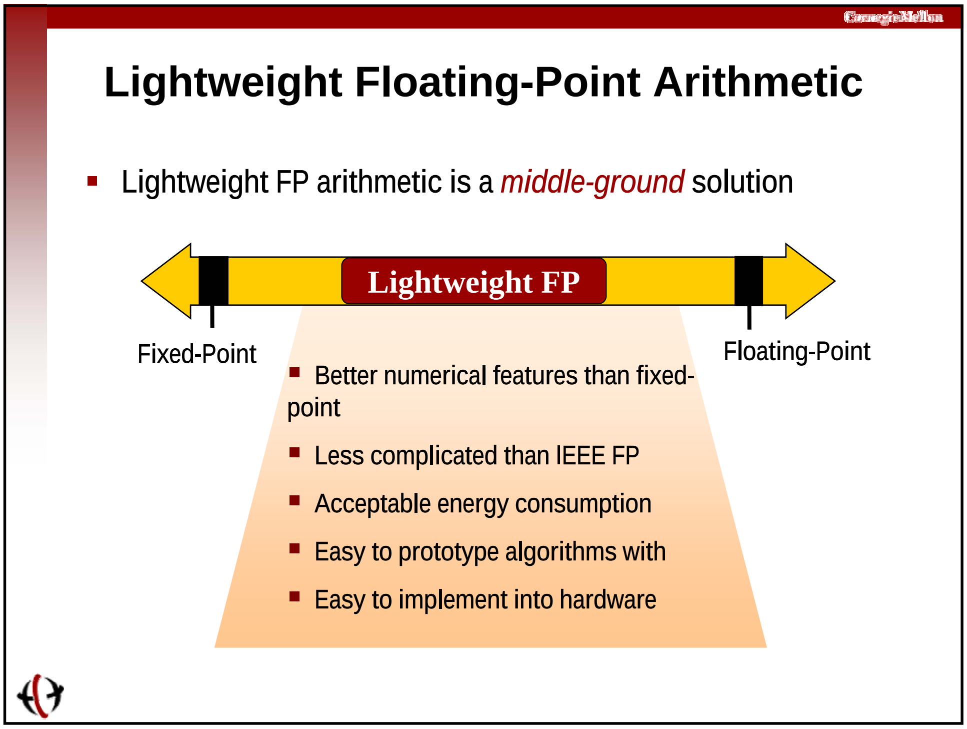
FP Formats and ops for *ordinary* numbers

Very small nums:denormals

Delicate rounding modes

……

We can work on each dimension



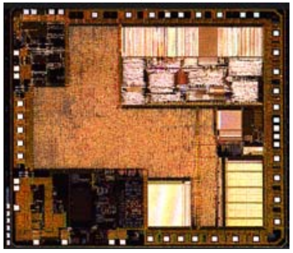
Lightweight Floating-Point Arithmetic

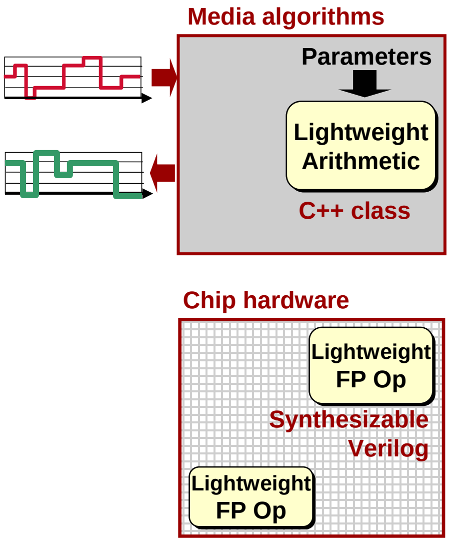
- Lightweight FP arithmetic is a *middle-ground* solution

Lightweight FP
Fixed-Point
Floating-Point

- Better numerical features than fixed-point
- Less complicated than IEEE FP
- Acceptable energy consumption
- Easy to prototype algorithms with
- Easy to implement into hardware

Software to Hardware Cycles





Media algorithms

Parameters

Lightweight Arithmetic

C++ class

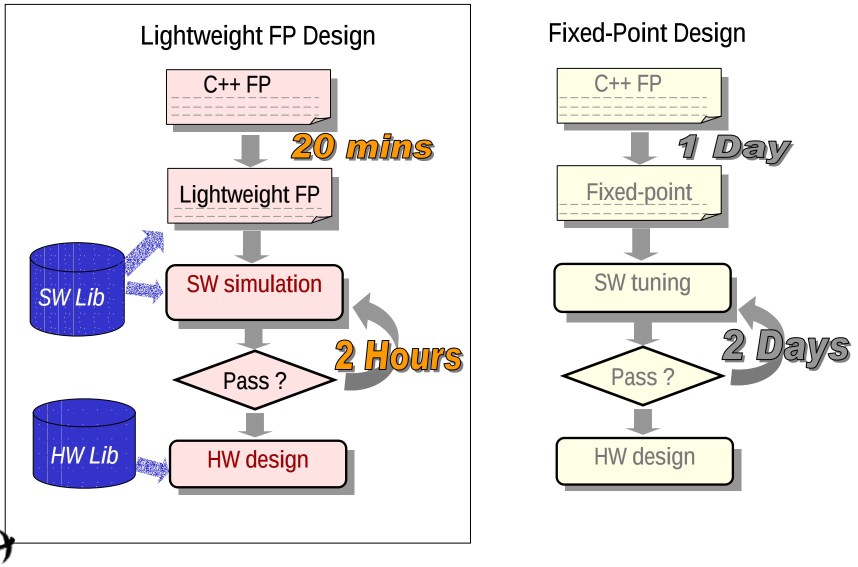
Chip hardware

Lightweight FP Op

Synthesizable Verilog

Lightweight FP Op

Design Flow Comparison



Lightweight FP Design

C++ FP

20 mins

Lightweight FP

SW Lib

SW simulation

2 Hours

Pass ?

HW Lib

HW design

Fixed-Point Design

C++ FP

1 Day

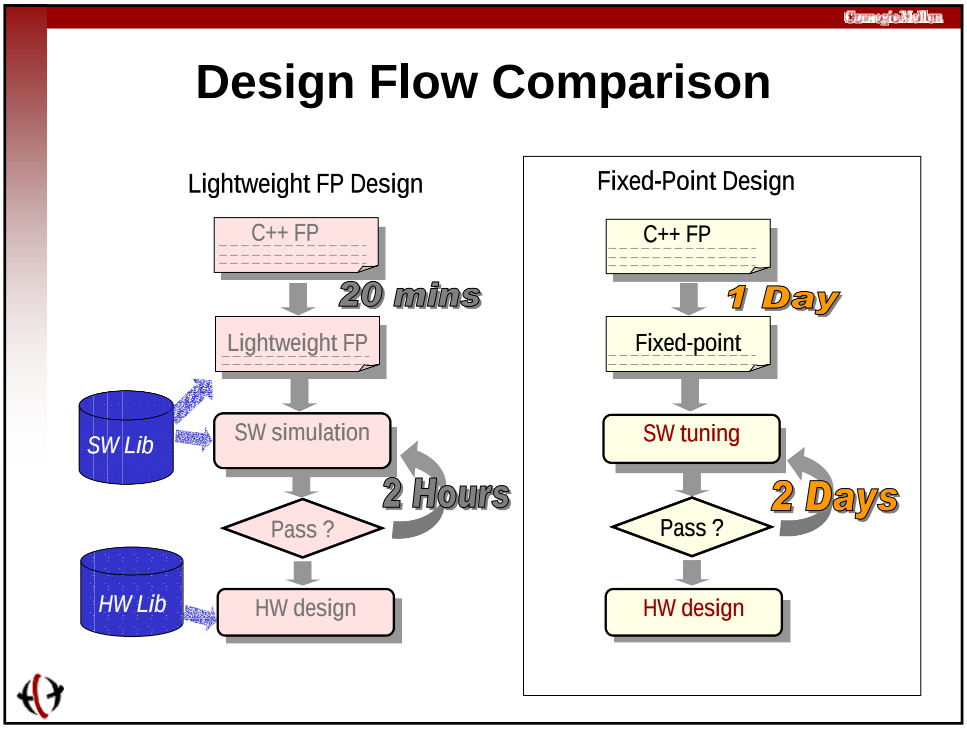
Fixed-point

SW tuning

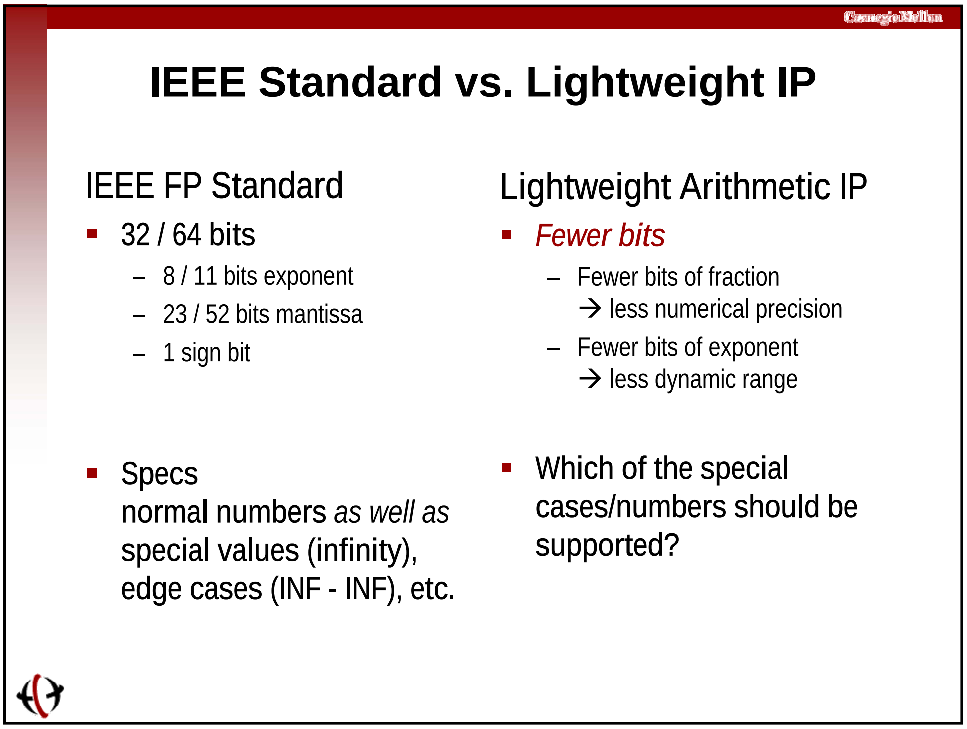
2 Days

Pass ?

HW design



Design Flow Comparison



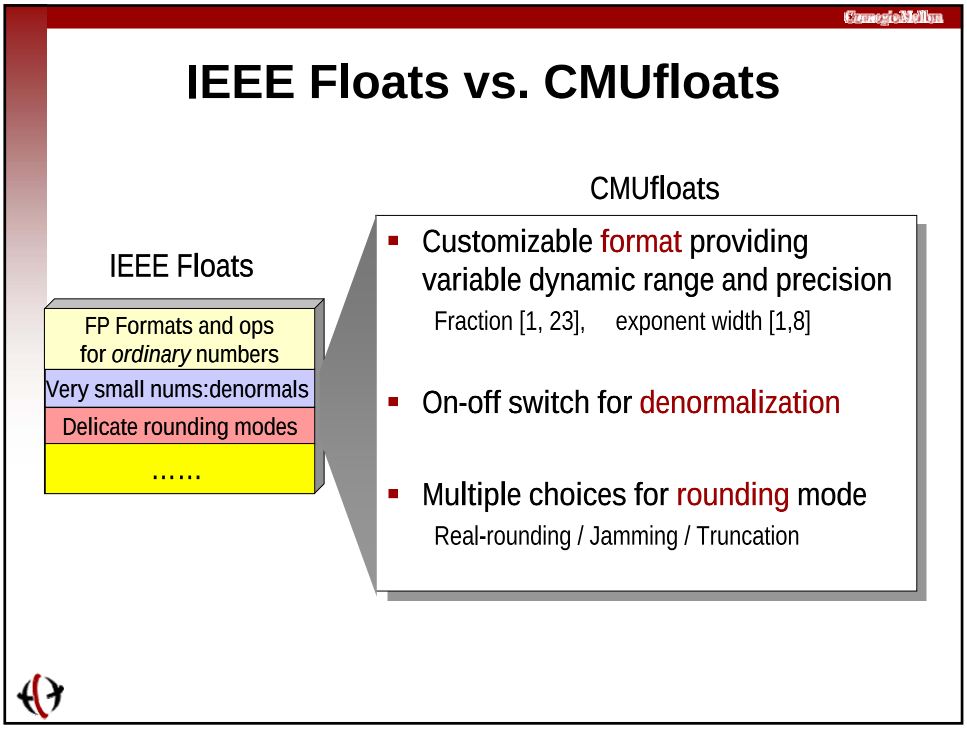
IEEE Standard vs. Lightweight IP

IEEE FP Standard

- 32 / 64 bits
  - 8 / 11 bits exponent
  - 23 / 52 bits mantissa
  - 1 sign bit
- Specs
  normal numbers *as well as*
  special values (infinity),
  edge cases (INF - INF), etc.

Lightweight Arithmetic IP

- *Fewer bits*
  - Fewer bits of fraction
    → less numerical precision
  - Fewer bits of exponent
    → less dynamic range
- Which of the special cases/numbers should be supported?



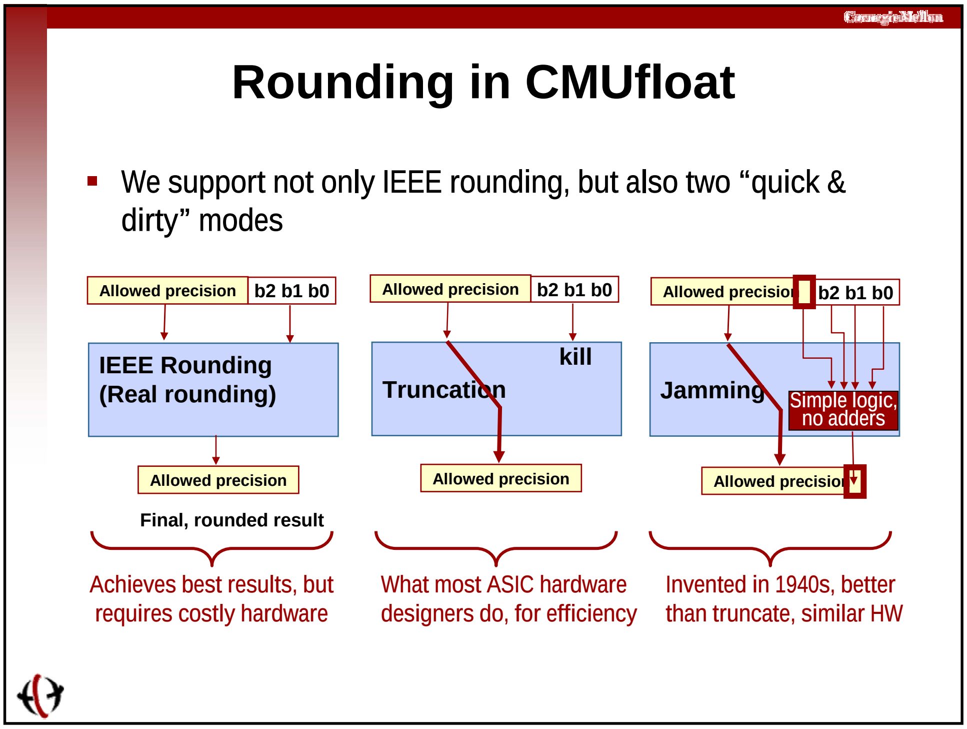
IEEE Floats vs. CMUfloats

IEEE Floats

FP Formats and ops
for *ordinary* numbers
Very small nums:denormals
Delicate rounding modes
……

CMUfloats

- Customizable format providing variable dynamic range and precision
  Fraction [1, 23], exponent width [1,8]
- On-off switch for denormalization
- Multiple choices for rounding mode
  Real-rounding / Jamming / Truncation



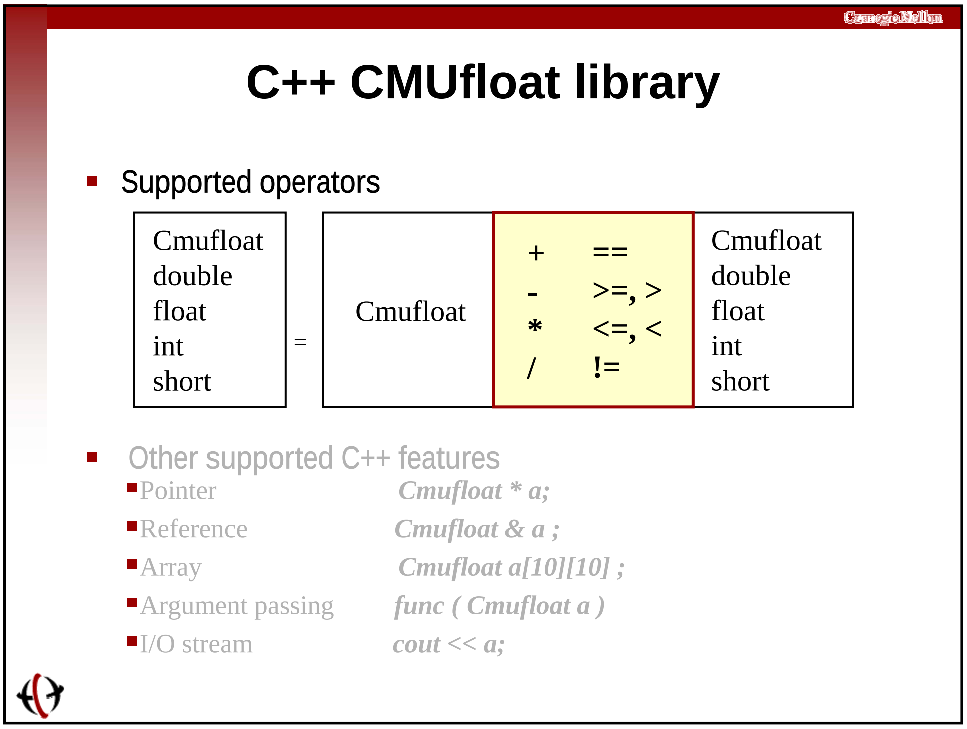
Rounding in CMUfloat

- We support not only IEEE rounding, but also two “quick & dirty” modes

Achieves best results, but requires costly hardware

What most ASIC hardware designers do, for efficiency

Invented in 1940s, better than truncate, similar HW



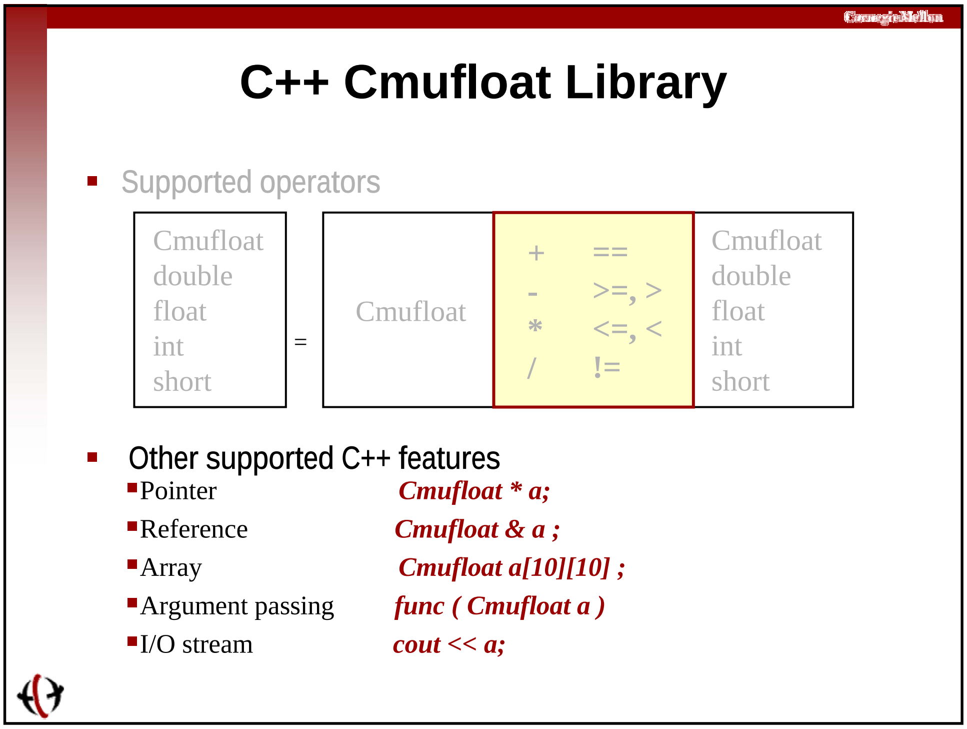
C++ CMUfloat library

- Supported operators

Cmufloat double float int short	=	Cmufloat	+ - * /	== >=, > <=, < !=	Cmufloat double float int short

- Other supported C++ features

Pointer	***Cmufloat * a;***
Reference	***Cmufloat & a ;***
Array	***Cmufloat a[10][10] ;***
Argument passing	***func (Cmufloat a)***
I/O stream	***cout << a;***



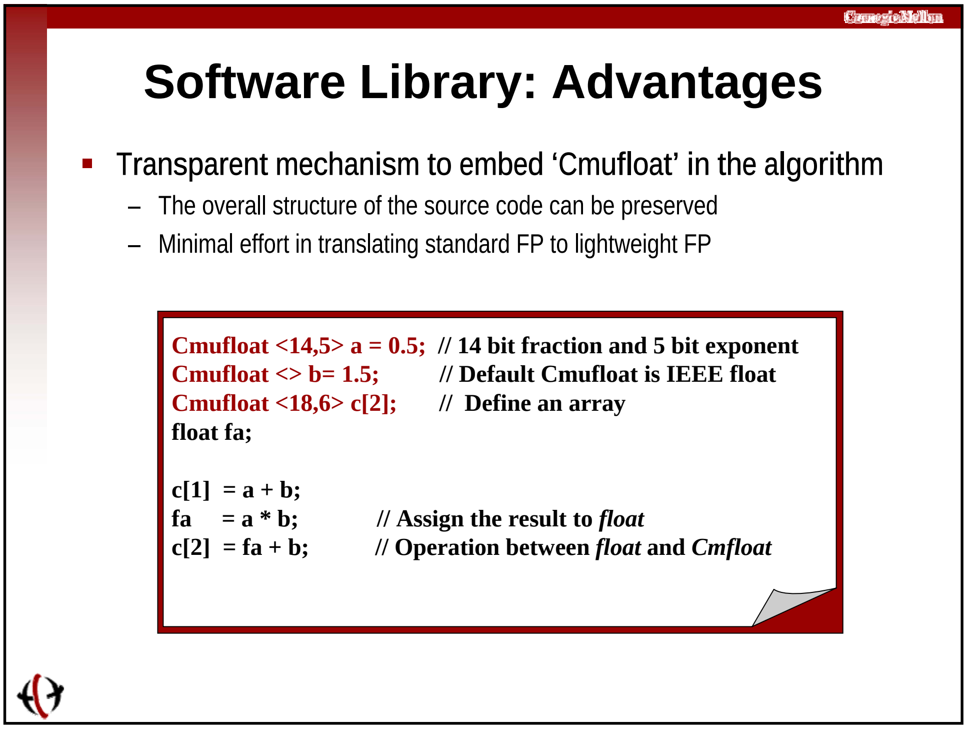
C++ Cmufloat Library

- Supported operators

Cmufloat double float int short	=	Cmufloat	+ - * /	== >=, > <=, < !=	Cmufloat double float int short

- Other supported C++ features

Pointer	***Cmufloat * a;***
Reference	***Cmufloat & a ;***
Array	***Cmufloat a[10][10] ;***
Argument passing	***func (Cmufloat a)***
I/O stream	***cout << a;***

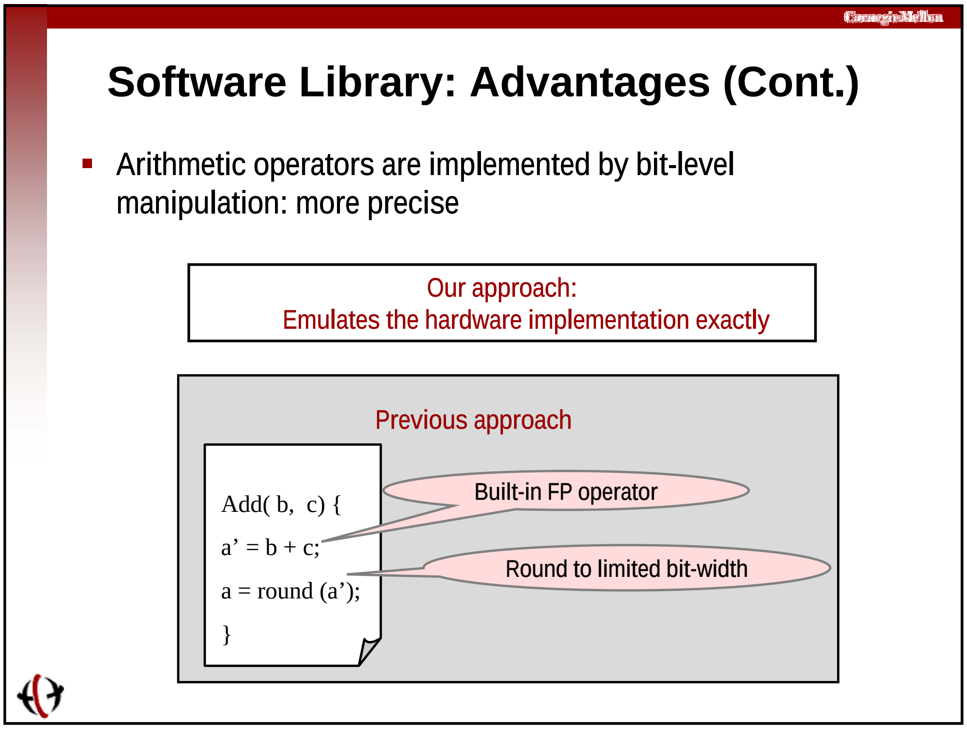


Software Library: Advantages

- Transparent mechanism to embed ‘Cmufloat’ in the algorithm
  - The overall structure of the source code can be preserved
  - Minimal effort in translating standard FP to lightweight FP

```
Cmufloat <14,5> a = 0.5;  // 14 bit fraction and 5 bit exponent
Cmufloat <> b= 1.5;         // Default Cmufloat is IEEE float
Cmufloat <18,6> c[2];       //  Define an array
float fa;

c[1]  = a + b;
fa     = a * b;            // Assign the result to float
c[2]  = fa + b;           // Operation between float and Cmfloat
```



Software Library: Advantages (Cont.)

- Arithmetic operators are implemented by bit-level manipulation: more precise

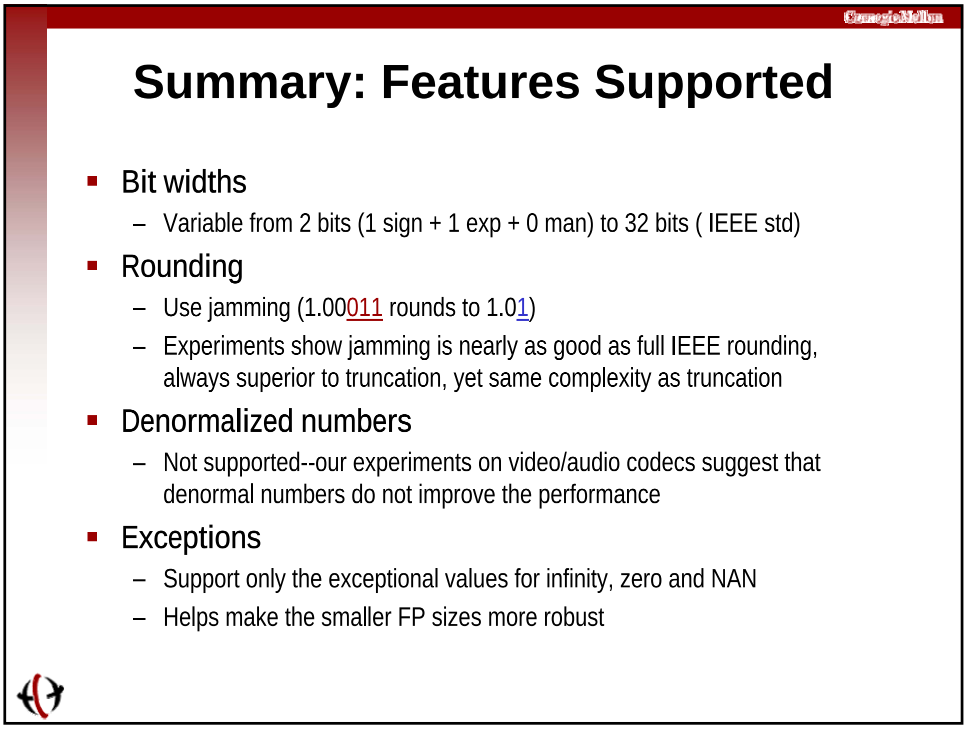
Our approach:
Emulates the hardware implementation exactly

Previous approach

```
Add( b,  c) {
a’ = b + c;
a = round (a’);
}
```

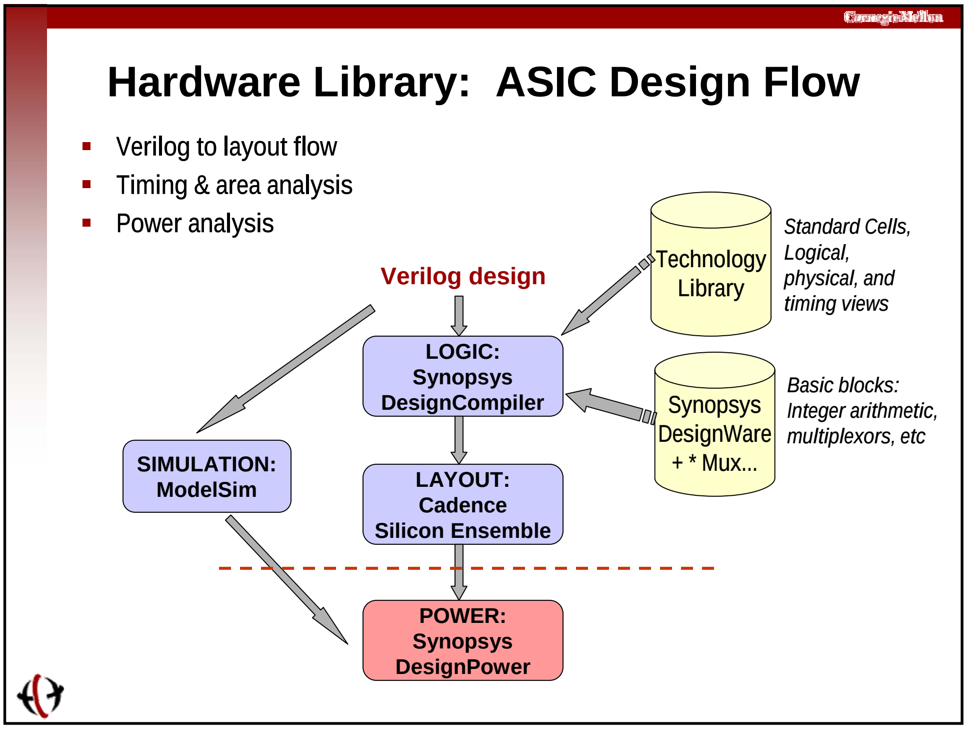
Built-in FP operator

Round to limited bit-width



Summary: Features Supported

- Bit widths
  - Variable from 2 bits (1 sign + 1 exp + 0 man) to 32 bits (IEEE std)
- Rounding
  - Use jamming (1.00011 rounds to 1.01)
  - Experiments show jamming is nearly as good as full IEEE rounding, always superior to truncation, yet same complexity as truncation
- Denormalized numbers
  - Not supported--our experiments on video/audio codecs suggest that denormal numbers do not improve the performance
- Exceptions
  - Support only the exceptional values for infinity, zero and NAN
  - Helps make the smaller FP sizes more robust



Hardware Library: ASIC Design Flow

- Verilog to layout flow
- Timing & area analysis
- Power analysis

Verilog design

LOGIC: Synopsys DesignCompiler

LAYOUT: Cadence Silicon Ensemble

SIMULATION: ModelSim

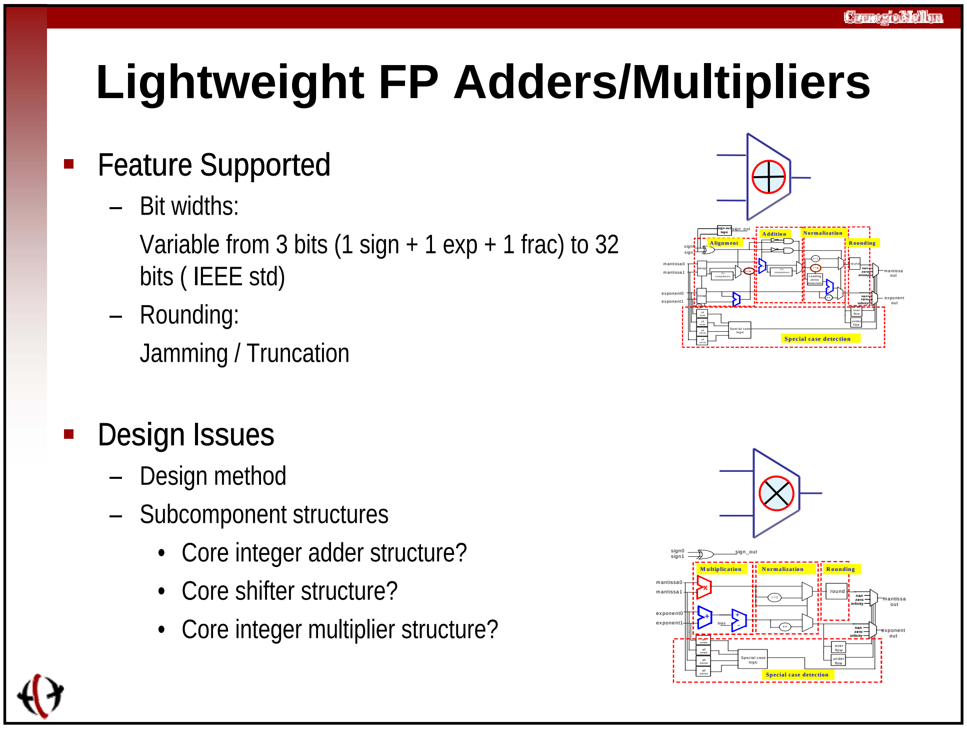
POWER: Synopsys DesignPower

Technology Library

Standard Cells, Logical, physical, and timing views

Synopsys DesignWare + * Mux...

Basic blocks: Integer arithmetic, multiplexors, etc



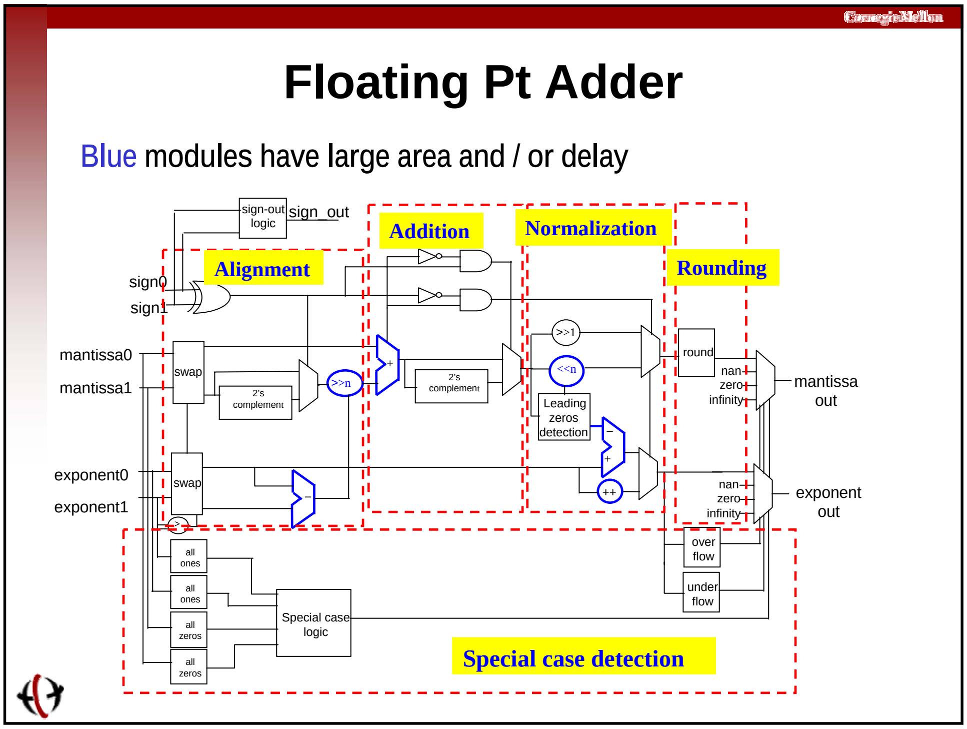
Lightweight FP Adders/Multipliers

- Feature Supported
  - Bit widths:

    Variable from 3 bits (1 sign + 1 exp + 1 frac) to 32 bits (IEEE std)
  - Rounding:

    Jamming / Truncation
- Design Issues
  - Design method
  - Subcomponent structures
    - Core integer adder structure?
    - Core shifter structure?
    - Core integer multiplier structure?

Alignment
Addition
Normalization
Rounding
Special case detection
mantissa out
exponent out
Multiplication
Normalization
Rounding
Special case detection
sign_out
mantissa out
exponent out

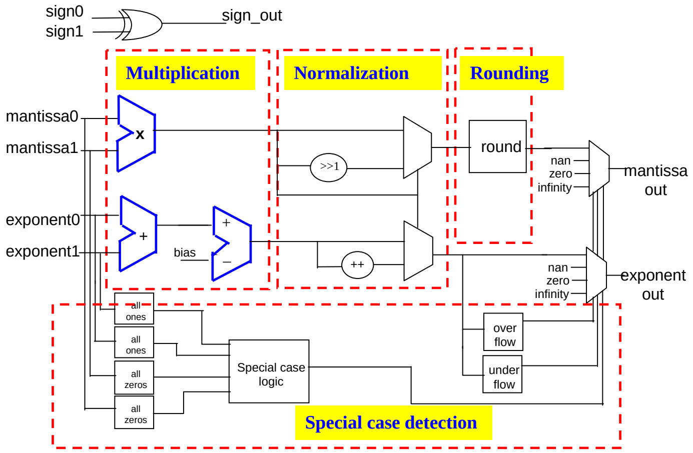


Floating Pt Adder

Blue modules have large area and / or delay

Floating Pt Multiplier

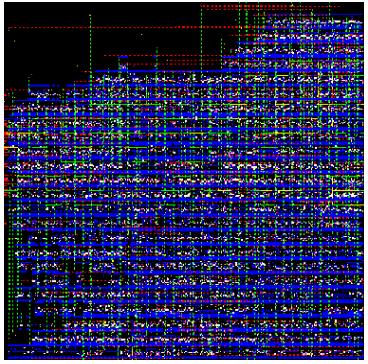
Blue modules have large area and / or delay



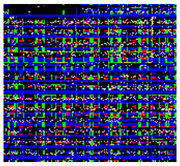
We see that the multiplier has less 'over-head' than the adder

Design Examples: Adders

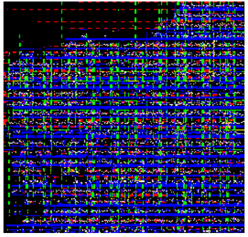
32-bit Floating-point



20-bit Fixed-pt



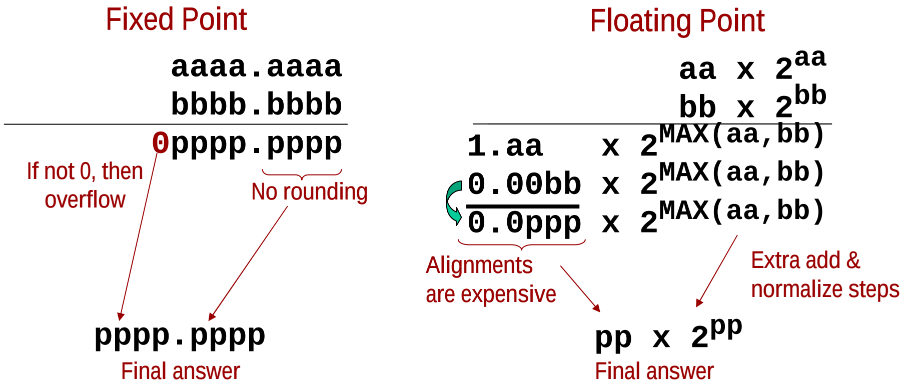
14-bit Floating-pt



	32-bit FP	20-bit FIX	14-bit FP
Area(um^2) - post layout	26634	4866	10096
Delay(ns) - post synthesys	48.95	2.44	25.77

Adder Analysis

Fixed-point adder is *very* simple; floating point adder is complex



Fixed Point

`aaaa.aaaa`

`bbbb.bbbb`

`0pppp.pppp`

If not 0, then overflow

No rounding

`pppp.pppp`

Final answer

Floating Point

$\mathtt{aa} \times 2^{\mathtt{aa}}$

$\mathtt{bb} \times 2^{\mathtt{bb}}$

$\mathtt{1.aa} \times 2^{\mathtt{MAX(aa,bb)}}$

$\mathtt{0.00bb} \times 2^{\mathtt{MAX(aa,bb)}}$

$\mathtt{0.0ppp} \times 2^{\mathtt{MAX(aa,bb)}}$

Alignments are expensive

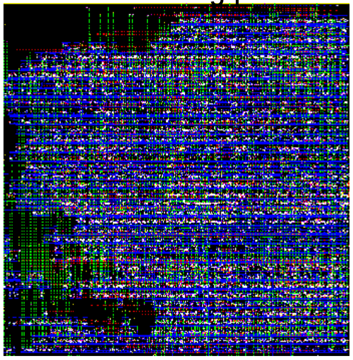
Extra add & normalize steps

$\mathtt{pp} \times 2^{\mathtt{pp}}$

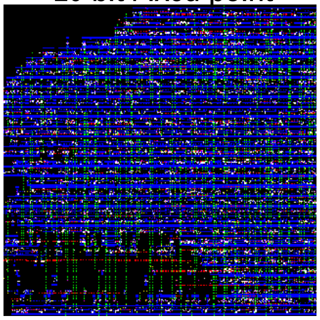
Final answer

Design Examples: Multipliers

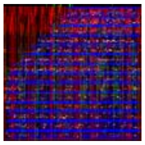
32-bit Floating-point



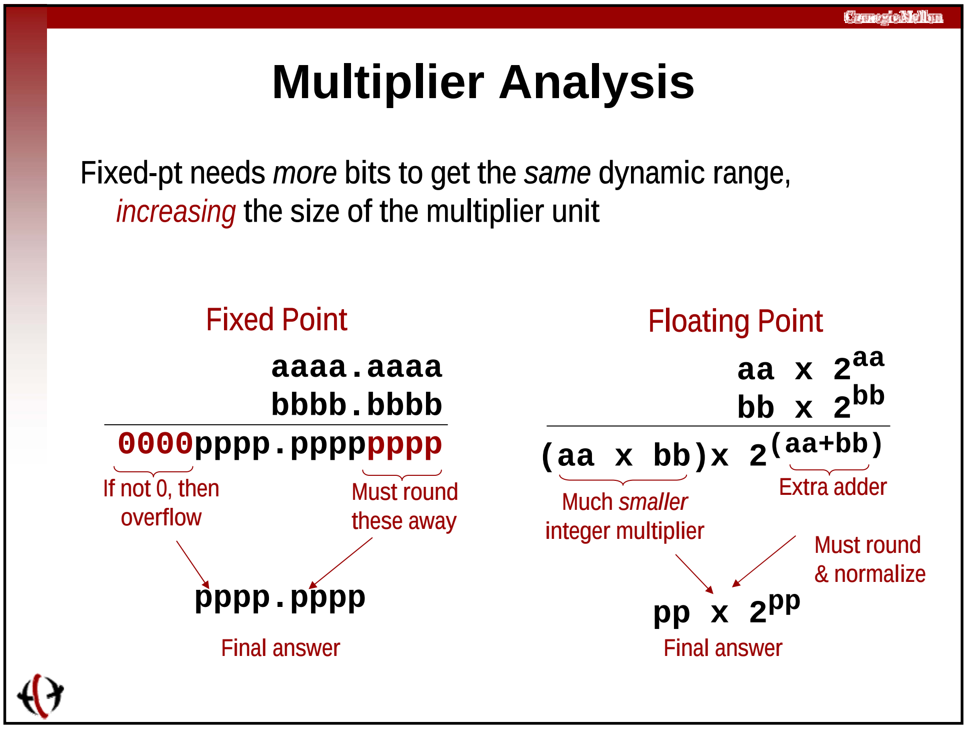
20-bit Fixed-point



14-bit Floating-pt



	32-bit FP	20-bit FIX	14-bit FP
Area(um^2) - post layout	60713	40738	8851
Delay(ns) - post synthesis	24.14	22.82	15.89



Multiplier Analysis

Fixed-pt needs *more* bits to get the *same* dynamic range, *increasing* the size of the multiplier unit

Fixed Point

$$\begin{array}{r} \texttt{aaaa.aaaa} \\ \texttt{bbbb.bbbb} \\ \hline \underbrace{\texttt{0000}}_{\text{If not 0, then overflow}}\texttt{pppp.pppp}\underbrace{\texttt{pppp}}_{\text{Must round these away}} \end{array}$$

$\texttt{pppp.pppp}$

Final answer

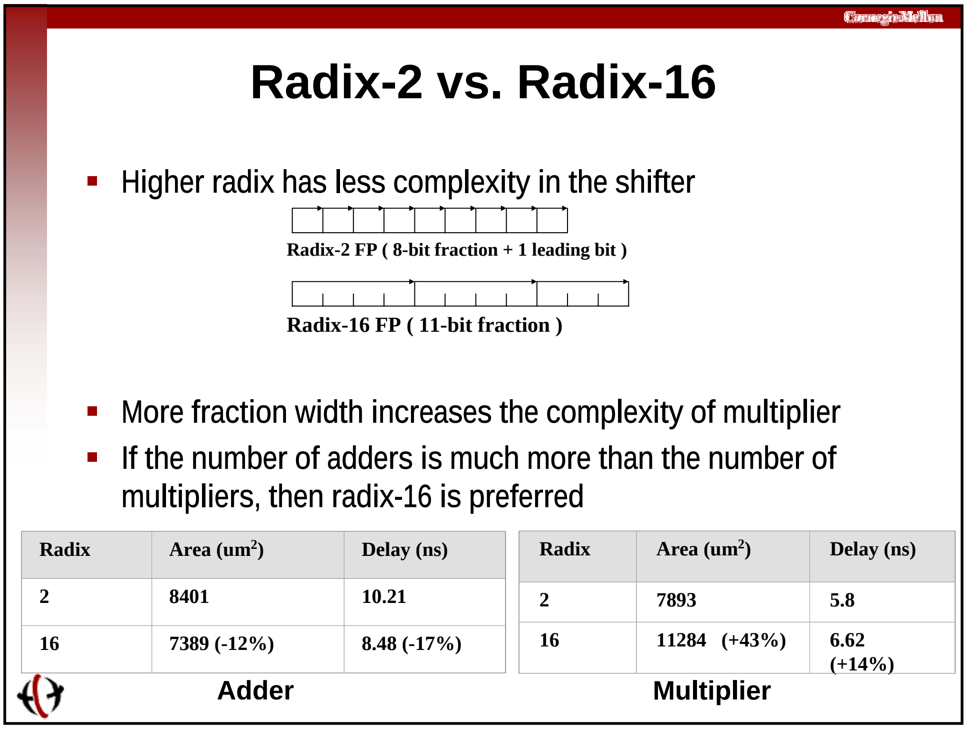
Floating Point

$$\begin{array}{r} aa \times 2^{aa} \\ bb \times 2^{bb} \\ \hline \underbrace{(aa \times bb)}_{\text{Much } \textit{smaller} \text{ integer multiplier}} \times 2^{\underbrace{(aa+bb)}_{\text{Extra adder}}} \end{array}$$

Must round & normalize

$pp \times 2^{pp}$

Final answer



Radix-2 vs. Radix-16

- Higher radix has less complexity in the shifter

Radix-2 FP (8-bit fraction + 1 leading bit)

Radix-16 FP (11-bit fraction)

- More fraction width increases the complexity of multiplier
- If the number of adders is much more than the number of multipliers, then radix-16 is preferred

Radix	Area (um^2)	Delay (ns)
2	8401	10.21
16	7389 (-12%)	8.48 (-17%)

Adder

Radix	Area (um^2)	Delay (ns)
2	7893	5.8
16	11284 (+43%)	6.62 (+14%)

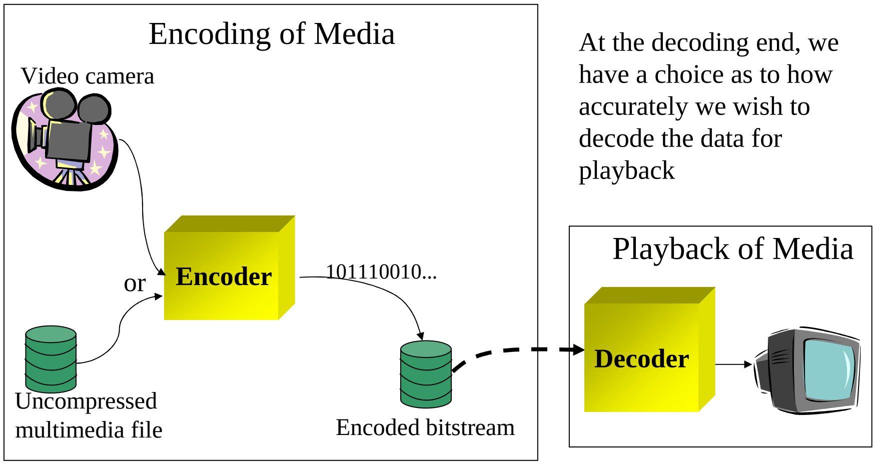
Multiplier

Power Analysis

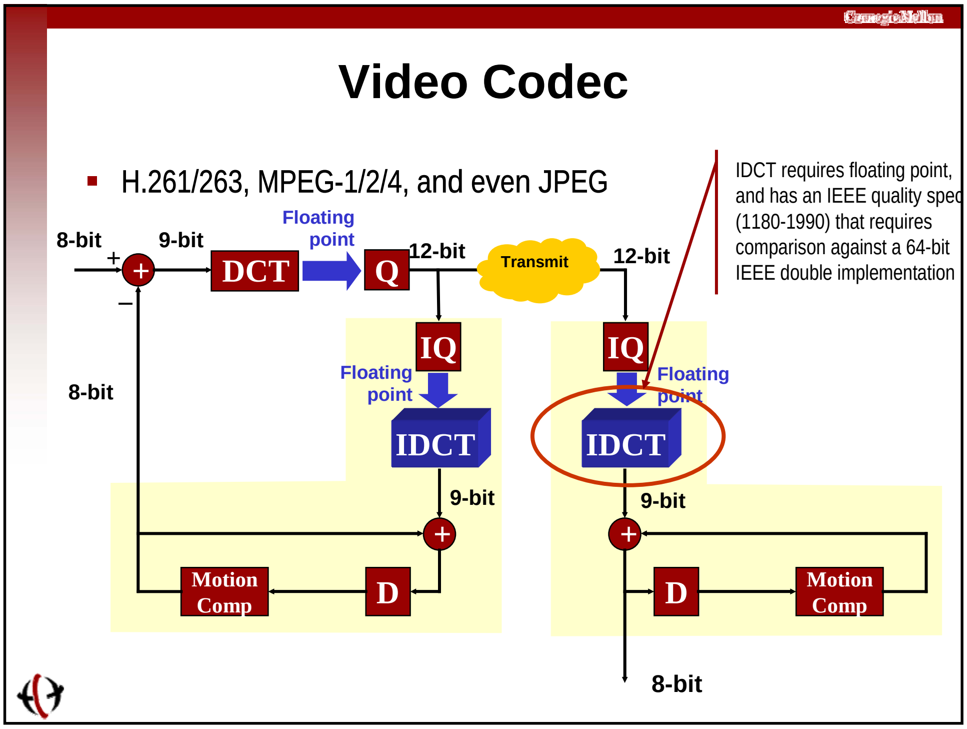
- IDCT in
  - 32-bit IEEE FP
  - 15-bit radix-16 lightweight FP
  - Fixed-point implementation
    - 12-bit accuracy for constants
    - Widest bit-width is 24 in the whole algorithm (not fine tuned)

Implementation	Area(um^2)	Delay(ns)	Power(mw)
IEEE FP	926810	111	1360
Lightweight FP	216236	46.75	143
Fixed-point	106598	36.11	110

Multimedia Encoding/Decoding



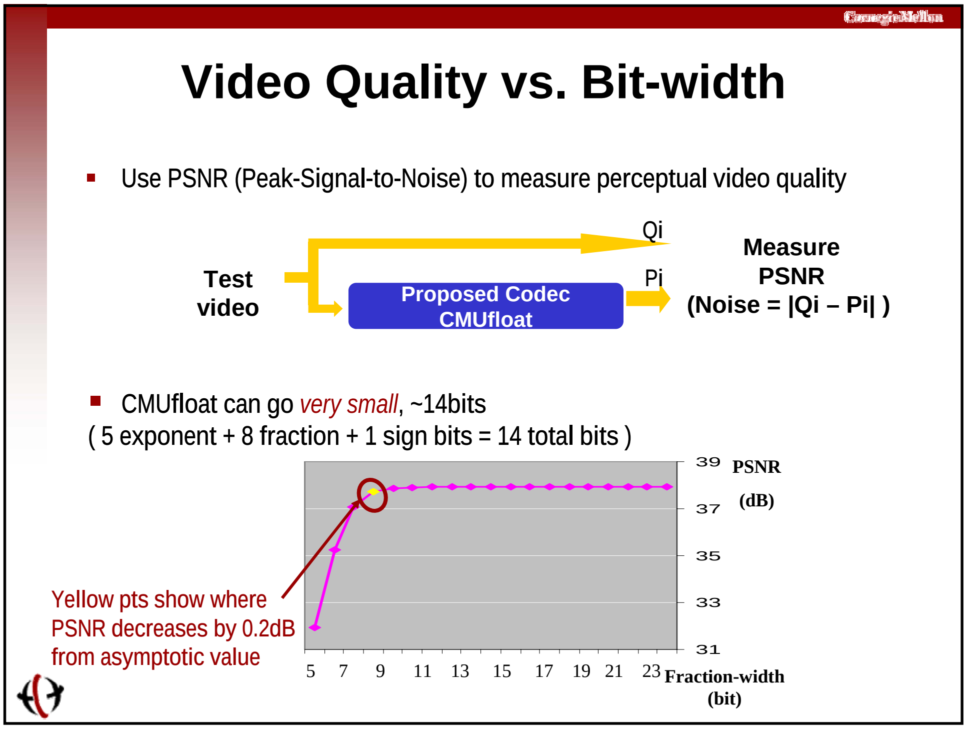
At the decoding end, we have a choice as to how accurately we wish to decode the data for playback



Video Codec

- H.261/263, MPEG-1/2/4, and even JPEG

IDCT requires floating point, and has an IEEE quality spec (1180-1990) that requires comparison against a 64-bit IEEE double implementation



Video Quality vs. Bit-width

- Use PSNR (Peak-Signal-to-Noise) to measure perceptual video quality

Test video

Proposed Codec CMUfloat

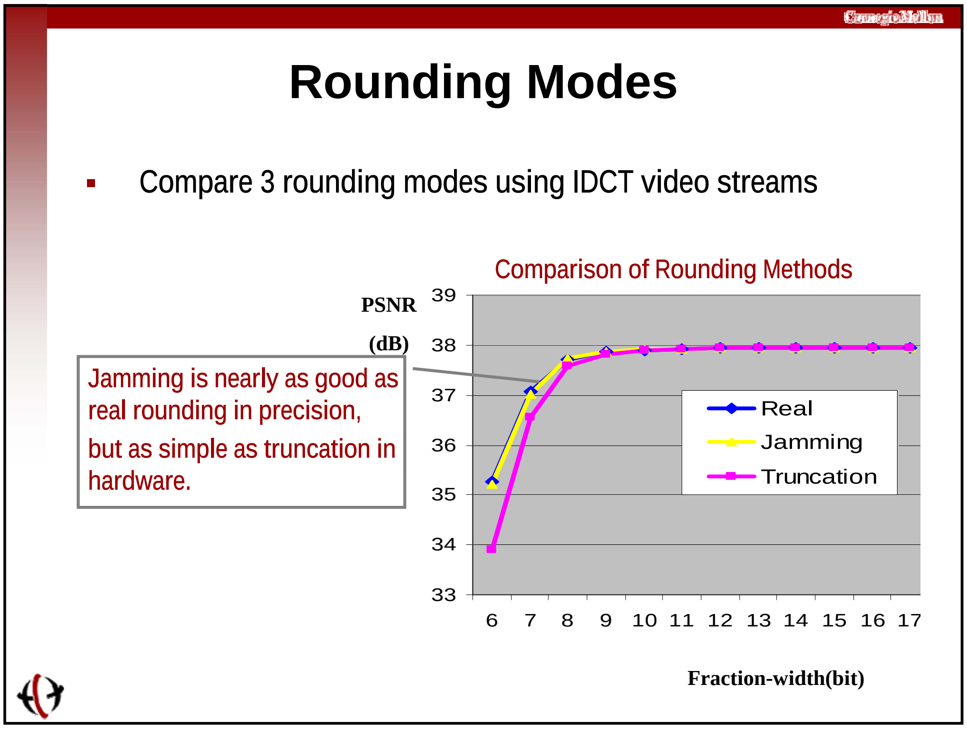
Qi

Pi

Measure PSNR (Noise = |Qi – Pi|)

- CMUfloat can go *very small*, ~14bits
(5 exponent + 8 fraction + 1 sign bits = 14 total bits)

Yellow pts show where PSNR decreases by 0.2dB from asymptotic value

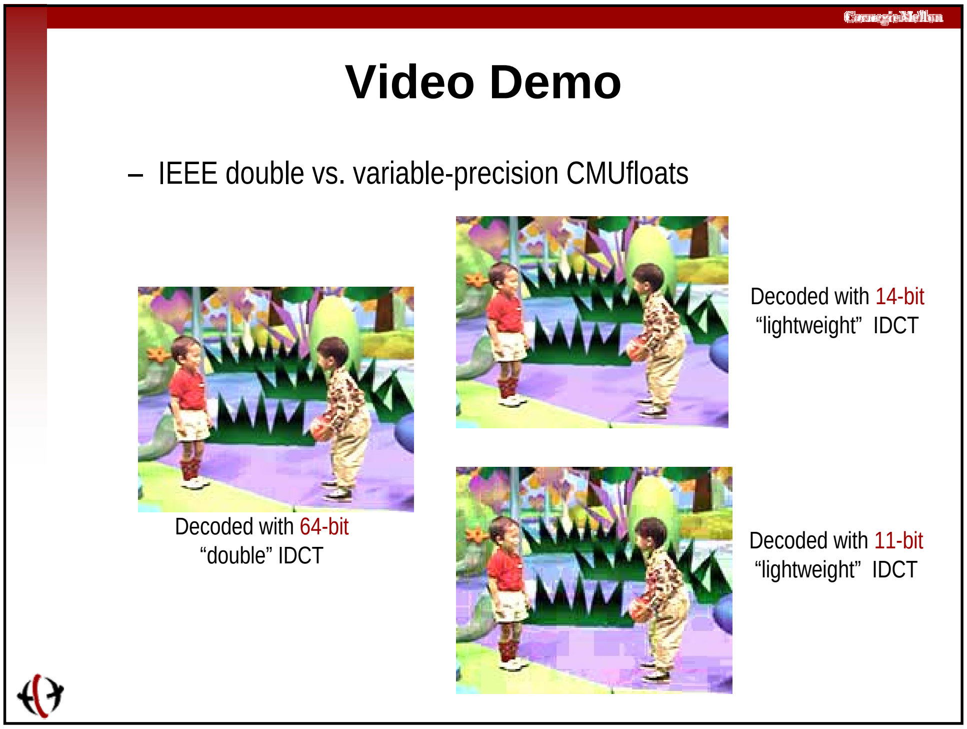


Rounding Modes

- Compare 3 rounding modes using IDCT video streams

Comparison of Rounding Methods

Jamming is nearly as good as real rounding in precision, but as simple as truncation in hardware.



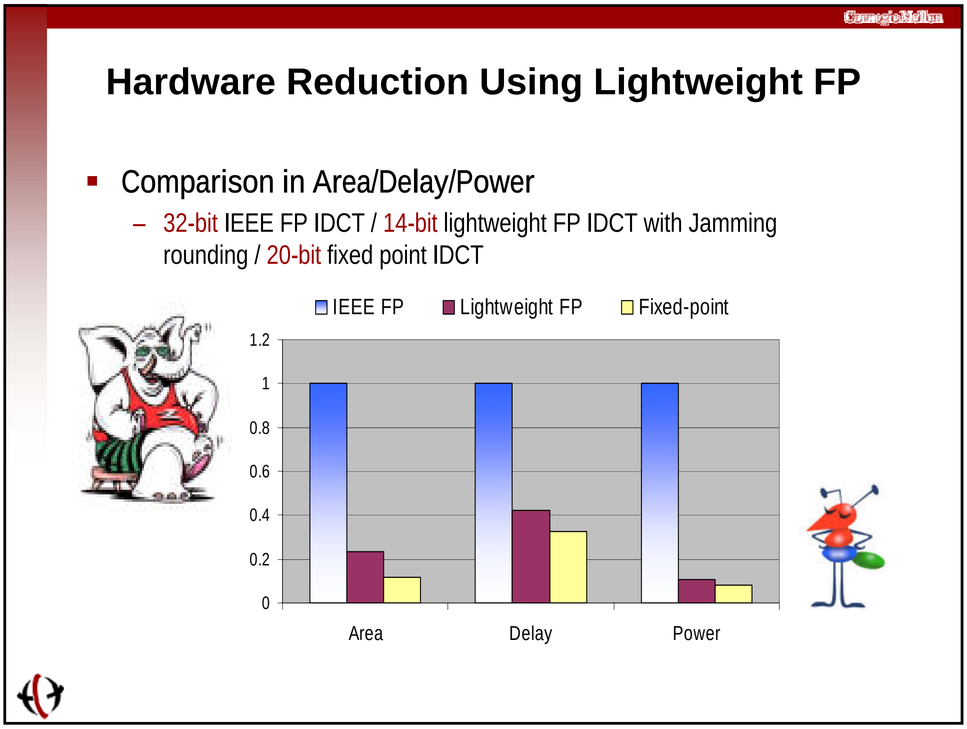
Video Demo

– IEEE double vs. variable-precision CMUfloats

Decoded with 64-bit "double" IDCT

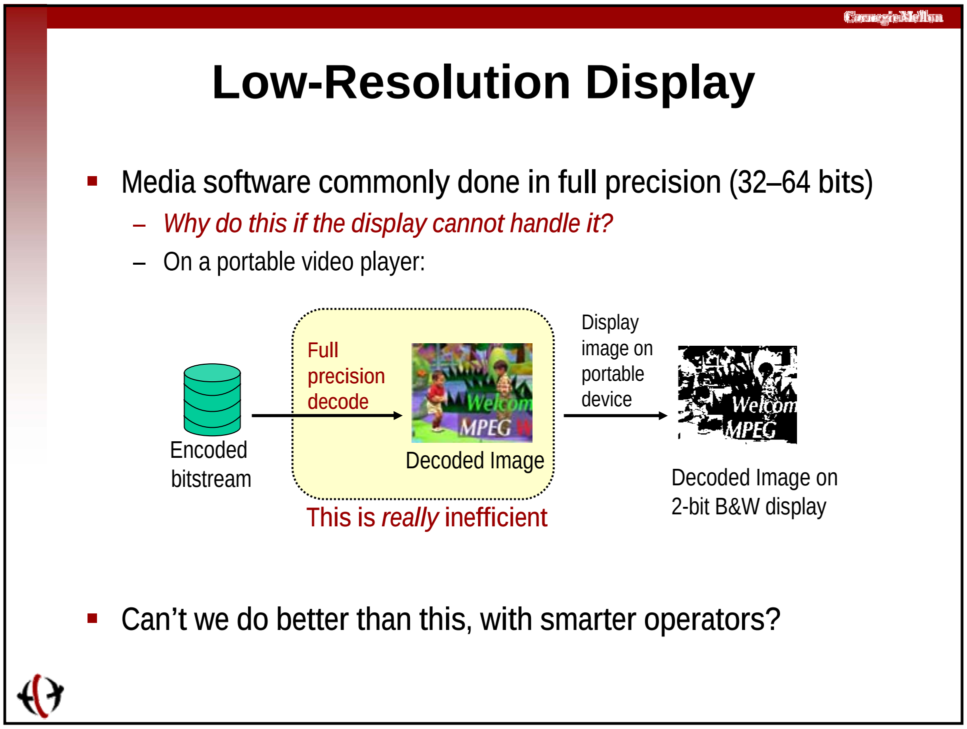
Decoded with 14-bit "lightweight" IDCT

Decoded with 11-bit "lightweight" IDCT



Hardware Reduction Using Lightweight FP

- Comparison in Area/Delay/Power
  - 32-bit IEEE FP IDCT / 14-bit lightweight FP IDCT with Jamming rounding / 20-bit fixed point IDCT



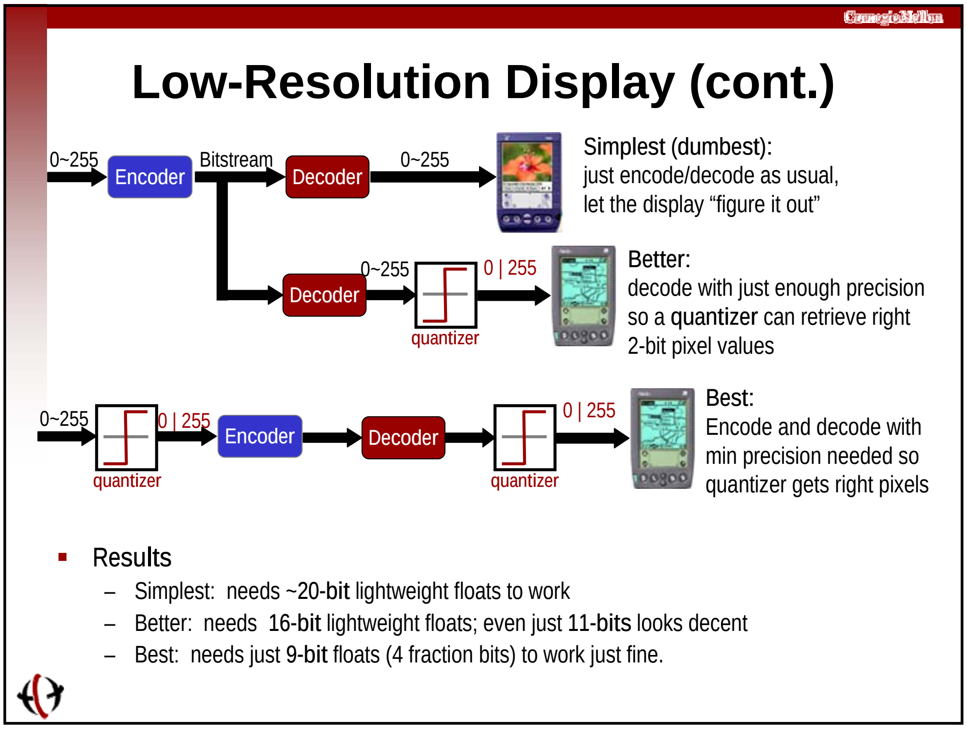
Low-Resolution Display

- Media software commonly done in full precision (32–64 bits)
  - *Why do this if the display cannot handle it?*
  - On a portable video player:

Encoded bitstream
Full precision decode
Decoded Image
Display image on portable device
Decoded Image on 2-bit B&W display
Welcom
MPEG

This is *really* inefficient

- Can't we do better than this, with smarter operators?



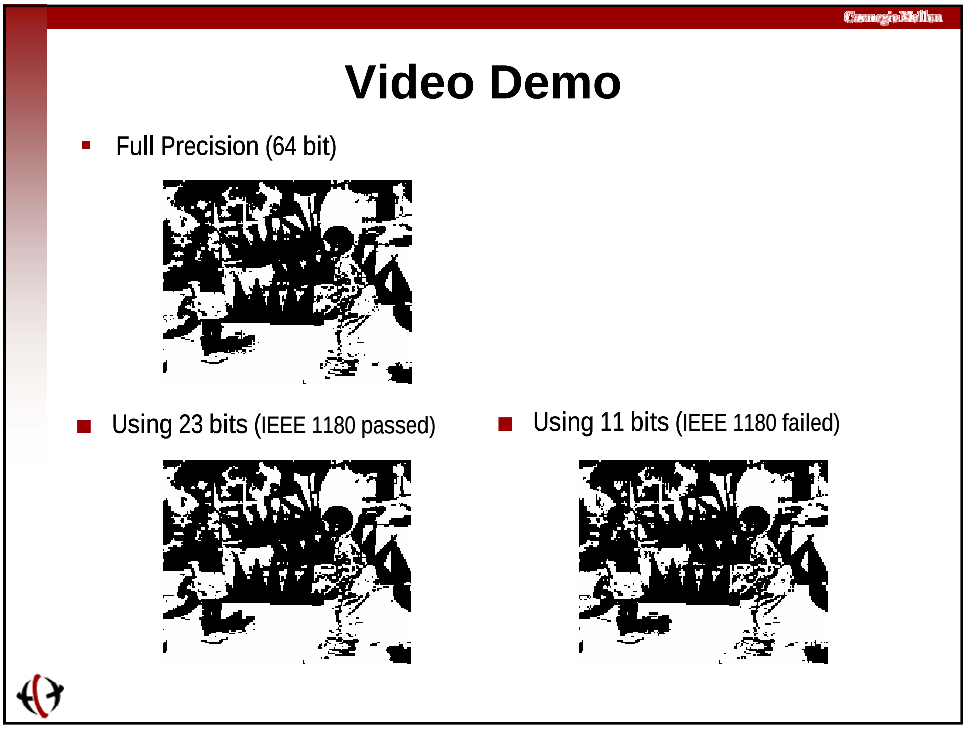
Low-Resolution Display (cont.)

Simplest (dumbest):
just encode/decode as usual,
let the display "figure it out"

Better:
decode with just enough precision
so a **quantizer** can retrieve right
2-bit pixel values

Best:
Encode and decode with
min precision needed so
quantizer gets right pixels

- Results
  - Simplest: needs ~**20-bit** lightweight floats to work
  - Better: needs **16-bit** lightweight floats; even just **11-bits** looks decent
  - Best: needs just **9-bit** floats (4 fraction bits) to work just fine.

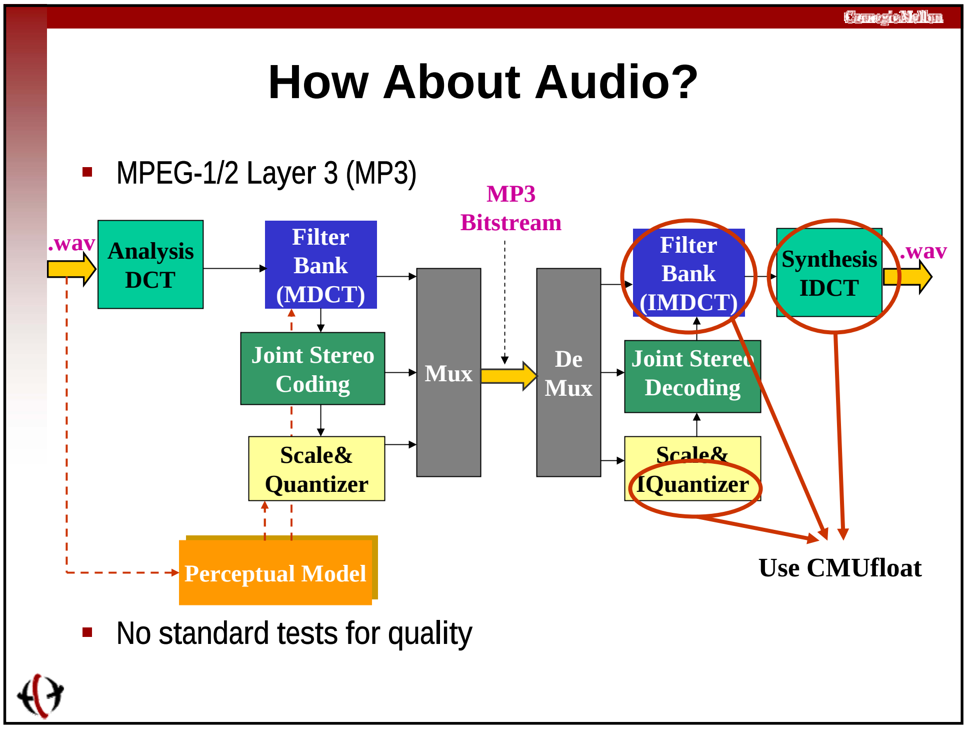


Video Demo

- Full Precision (64 bit)

- Using 23 bits (IEEE 1180 passed)

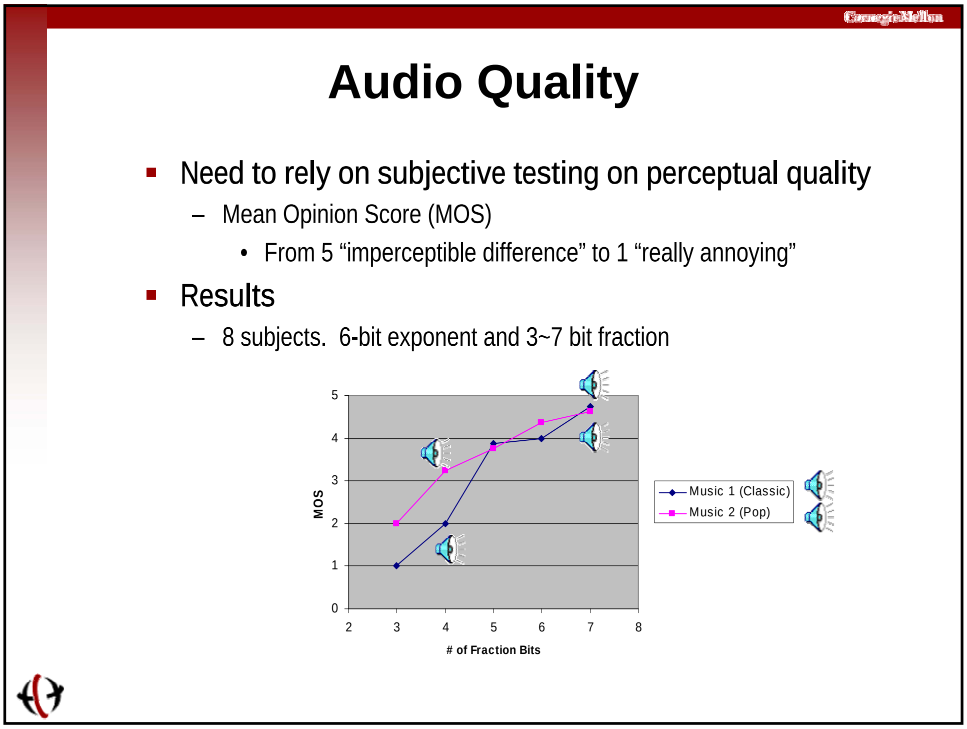
- Using 11 bits (IEEE 1180 failed)



How About Audio?

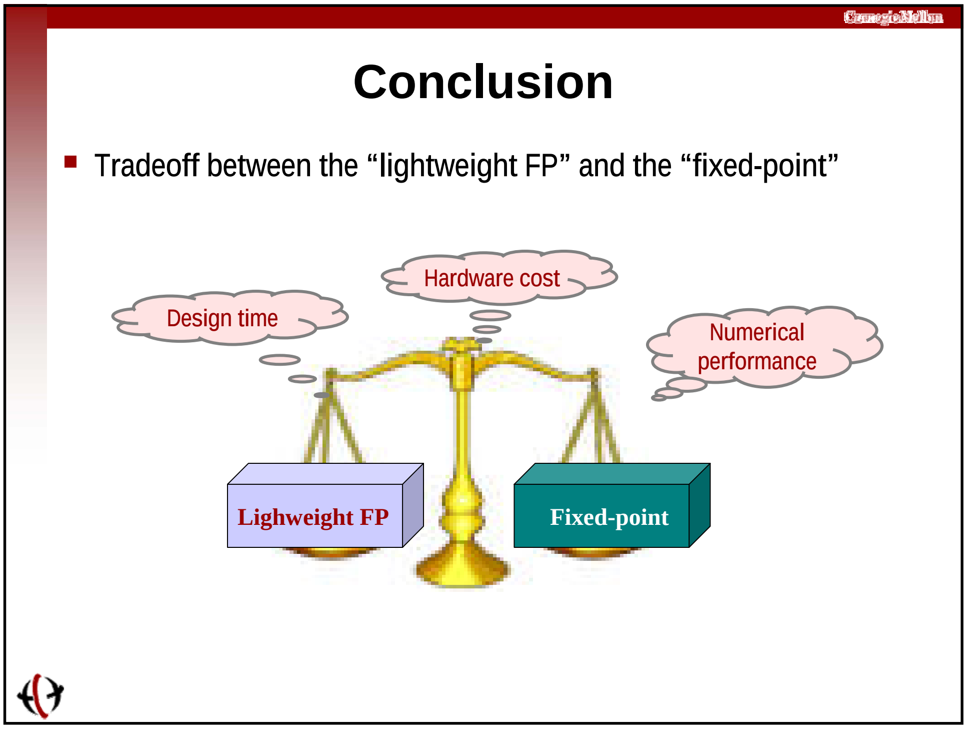
- MPEG-1/2 Layer 3 (MP3)

- No standard tests for quality



Audio Quality

- Need to rely on subjective testing on perceptual quality
  - Mean Opinion Score (MOS)
    - From 5 “imperceptible difference” to 1 “really annoying”
- Results
  - 8 subjects. 6-bit exponent and 3~7 bit fraction



Conclusion

- Tradeoff between the “lightweight FP” and the “fixed-point”

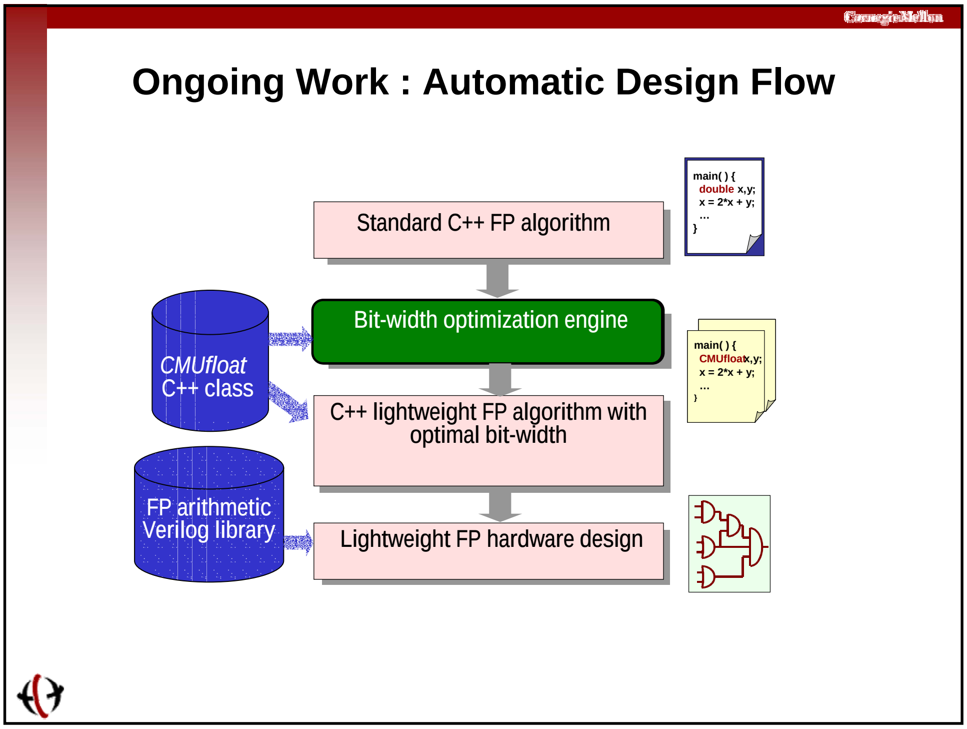
Design time

Hardware cost

Numerical performance

Lighweight FP

Fixed-point



Ongoing Work : Automatic Design Flow

Standard C++ FP algorithm

```
main( ) {
  double x,y;
  x = 2*x + y;
  …
}
```

Bit-width optimization engine

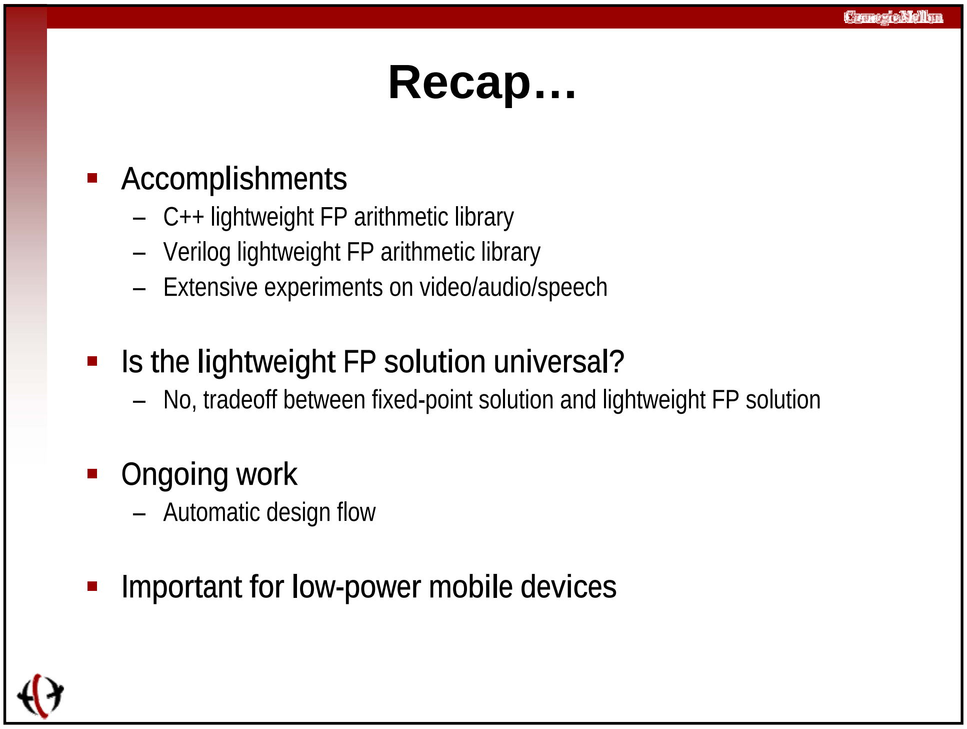
CMUfloat C++ class

```
main( ) {
  CMUfloat x,y;
  x = 2*x + y;
  …
}
```

C++ lightweight FP algorithm with optimal bit-width

FP arithmetic Verilog library

Lightweight FP hardware design



Recap…

- Accomplishments
  - C++ lightweight FP arithmetic library
  - Verilog lightweight FP arithmetic library
  - Extensive experiments on video/audio/speech
- Is the lightweight FP solution universal?
  - No, tradeoff between fixed-point solution and lightweight FP solution
- Ongoing work
  - Automatic design flow
- Important for low-power mobile devices

Advanced Multimedia Processing Lab

Please visit us at:

http://amp.ece.cmu.edu