Today

- Digital signal processors
  - VLIW
  - SHARC details
- Quick look at audio processing

Digital Signal Processors

- **Microcontrollers are optimized for control-intensive apps**
  - **Average general-purpose application branches every seven instructions**
  - **Branches often not very predictable**
  - **Memory accesses often not very predictable**
- **DSPs are optimized for math, loops, and data movement**
  - **Both fixed-point and floating-point math**
  - **Fast loop operations for simple loop structures**
  - **Lots of I/O**
  - **Instructions and memory accesses very predictable**

Important DSPs

- **Texas Instruments**
  - **TMS320C2000, TMS320C5000, and TMS320C6000**
- **Motorola**
  - **StarCore: DSP56300, DSP56800, and MSC8100**
- **Agere Systems**
  - **DSP16000 series**
- **Analog Devices**
  - **SHARC: ADSP-2100 and ADSP-21000**

At the low end…

- DSP: All key arithmetic ops in 1 cycle
- GPP: Often some math (multiply at least) is multiple-cycle

- DSP: Support for 8 and 16 bit quantities as both integers and fractions
- GPP: Fixed word size, integer only

- DSP: HW support for managing numerical fidelity
  - Saturation, flexible rounding, etc.
- GPP: These are often implemented in SW

At the high end…

- DSP: Up to 8 arithmetic units
- GPP: 1-3 arithmetic units

- DSP: Highly specialized functional units
  - Multiply and accumulate, Viterbi, etc.
- GPP: General-purpose functional units
  - Integer, floating point, etc.

- DSP: Very limited use of dynamic features
  - Branch predication, superscalar, etc.
- GPP: Extensive use of dynamic features

More CPU vs. DSP

- DSPs are Harvard architecture even at the high end
  - No high end CPU is Harvard architecture
- DSPs offer better cache control
  - Lockable cache regions
  - Cache can be turned into scratchpad RAM
    - Scratchpad == explicitly addressable fast RAM
- DSP weaknesses
  - Not easy to program by hand, compilers can be flaky
  - Poor operating system support
  - Not good at executing control-intensive code

More CPU vs. DSP

- Many embedded systems contain
  - One or more MCUs
  - One or more DSPs
- Let each kind of processor run the kind of code it is good at

SHARC

- Medium-performance DSP architecture
- Similarities to MCF52233
  - Separate instruction and data memories
  - Some pipelining (3 stage vs. 4)
- SHARC is more CISC than ColdFire
  - CISC main idea
    - Give people complex instructions that match what they are trying to do
    - This gives good performance and high code density
  - SHARC
    - Instructions are highly specialized for DSP

Quick VLIW Intro

- VLIW == Very Long Instruction Word
- Aggressive superscalar, out-of-order processors like P4 and Athlon
  - Single operation per instruction
  - Get high IPC through superscalar and out-of-order execution
  - Requires lots of logic (and energy) to detect and avoid problematic dependencies
- VLIW
  - Dependencies detected and avoided at compile time
  - VLIW can get high IPC with simpler HW
  - Compiler technology is difficult
  - Also, compiler becomes very sensitive to the architectural details and program structure

More SHARC Stuff

- Supports saturating ALU operations
- Can issue some computations in parallel
  - Dual add-subtract
  - Multiplication and dual add/subtract
  - Floating-point multiply and ALU operation
- Example SHARC instruction:
  - `R6 = R0*R4, R9 = R8 + R12, R10 = R8 - R12;`

Parallelism Example

- **We want to compute:**
  - if (a>b) y = c-d; else y = c+d;
  - **Strategy: Compute both results in parallel and then pick the right one**

```
! Load values (DM == data memory)
R1=DM(_a); R2=DM(_b);
R3=DM(_c); R4=DM(_d);
! Compute both sum and difference
R12 = R2+R4, R0 = R2-R4;
! Choose which one to save
COMP(R1,R2);
IF LE R0=R12;
DM(_y) = R0 ! Write to y
```

SHARC Addressing

- **Immediate value**
  - `R0 = DM(0x20000000);`
- **Direct load**
  - `R0 = DM(_a);  ! Loads contents of _a`
- **Direct store**
  - `DM(_a)= R0;  ! Stores R0 at _a`
- **Post-modify with update**
  - Used to sweep through a buffer
  - I register holds base address
  - M register/immediate holds modifier value
  - `R0 = DM(I3,M3)  ! Load`
  - `DM(I2,1) = R1  ! Store`

Data in Program Memory

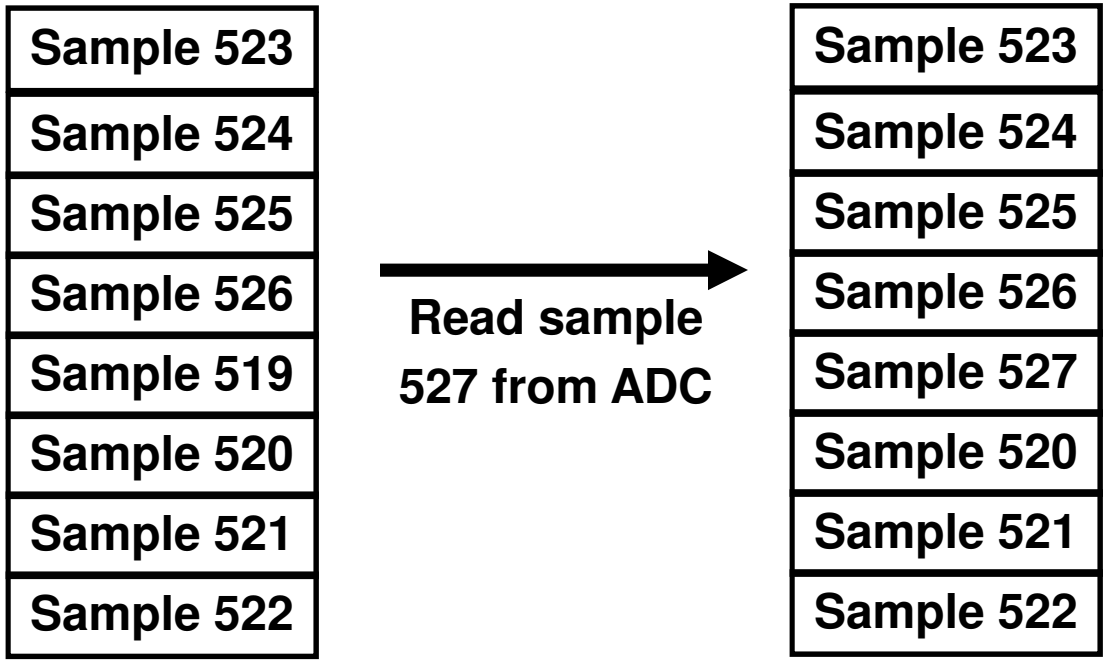
- Can put constant data in program memory to read two values per cycle:

```
F0 = DM(M0,I0), F1 = PM(M8,I9);
```

- Compiler allows programmer to control which memory values are stored in

Circular Buffers

- Fundamental data structure for DSP
- New sample always overwrites oldest sample



SHARC Circular Buffers

- **Uses special Data Address Generator registers:**
  - **B register is buffer base address**
  - **L register is buffer size**
  - **I, M registers in post-modify mode**
  - **I is automatically wrapped around the circular buffer when it reaches B+L**

SHARC Zero Overhead Loop

- **No cost for jumping back to start of loop**
  - **Hardware decrements counter, compares, then jumps back**


```
    LCNTR=30, DO L UNTIL LCE;
    R0=DM(I0,M0), F2=PM(I8,M8);
    R1=R0-R15;
L:  F4=F2+F3;
```

- **Nested loops also handled**
  - **HW provides a 6-deep loop counter stack**

FIR in Detail

1. Obtain sample from ADC, generate interrupt
2. Move the sample into the input circular buffer
3. Update the pointer for the circular buffer
4. Zero the accumulator
5. Loop through all coefficients
    1. Fetch coefficient from coefficient circular buffer
    2. Update pointer to coefficient circular buffer
    3. Fetch sample from input circular buffer
    4. Update the pointer to the input circular buffer
    5. Multiply coefficient and sample
    6. Add result to accumulator
6. Move output sample to a holding buffer
7. Move output sample from holding buffer to DAC

FIR Inner Loop in C

```
int fir_inner (void)
{
  int i, f;
  for (i=0, f=0; i<N; i++)
     f = f + c[i]*x[i];
  return f;
}
```

FIR Inner in ColdFire

```
fir_inner:
  link      a6,#0
  moveq     #0,d2
  moveq     #0,d0
  lea       _x,a1
  lea       _c,a0
  addq.l    #1,d2
  move.l    (a1)+,d1
  muls.l    (a0)+,d1
  add.l     d1,d0
  cmpi.l    #10,d2
  blt.s     *-16
  unlk      a6
  rts
```

FIR Inner in SHARC

```
! loop setup
I0=a;              ! I0 points to a[0]
M0=1;              ! set up increment
I8=b;              ! I8 points to b[0]
M8=1;              ! set up postincrement mode
! loop body
LCNTR=N, DO loopend UNTIL LCE;
R1=DM(I0,M0), R2=PM(I8,M8);
R8=R1*R2;
loopend:
R12=R12+R8;
```

DSP C Compilers

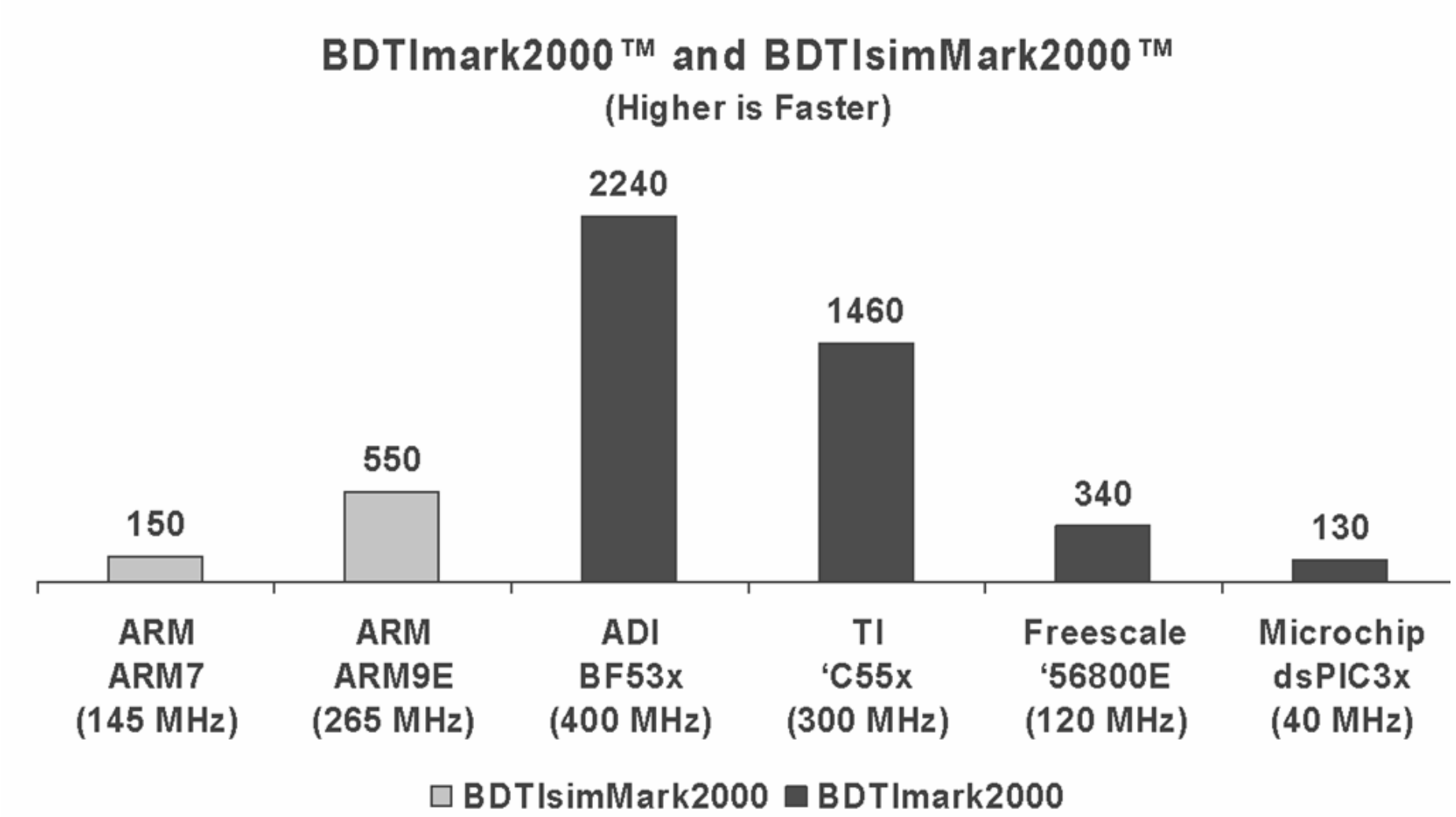
- Most of the compiler is the same as for standard architectures
  - Lexer, parser, type checker
  - IR generator
  - High-level optimizations
    - CSE, constant folding and propagation, loop unrolling
- Target-dependent optimizations are different
  - Software pipelining
  - Instruction scheduling
  - Peephole optimizations
  - Register allocation
- DSP compilers are typically very sensitive to issues like arrays vs. pointers

A few SHARCs

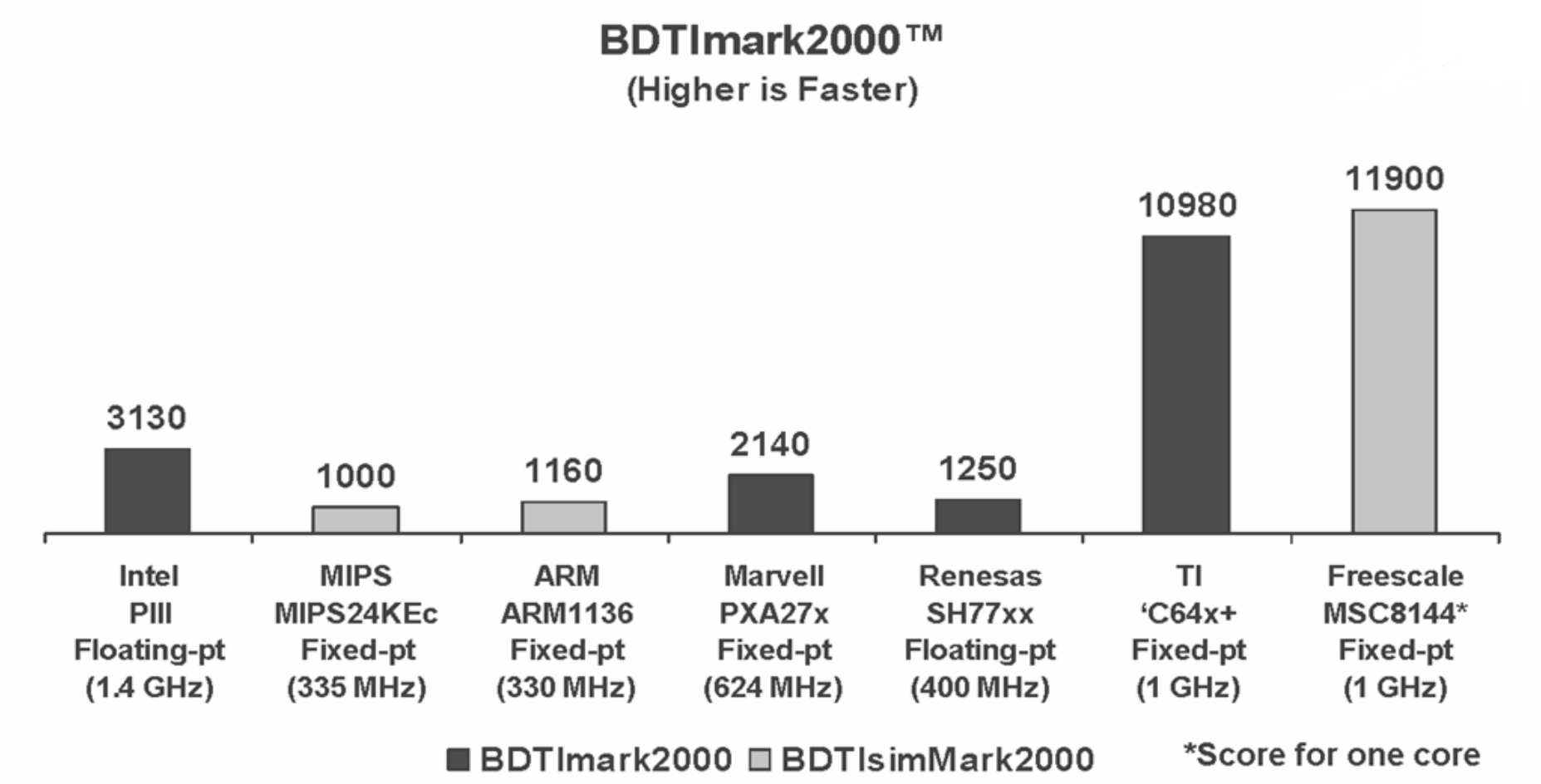
$12 **$60**

	ADSP-21261	**ADSP-21262 ADSP-21266**	**ADSP-21375**	**ADSP-21469**
Clock Cycle	150 MHz	200 MHz	266 MHz	450 MHz
Instruction Cycle Time	6.67 ns	5 ns	3.75 ns	2.22 ns
MFLOPS Sustained	600 MFLOPS	800 MFLOPS	1064 MFLOPS	1800 MFLOPS
MFLOPS Peak	900 MFLOPS	1200 MFLOPS	1596 MFLOPS	2700 MFLOPS
1024 Point Complex FFT (Radix 4, with bit reversal)	61.3 µs	46 µs	34.5 µs	20.4 µs
FIR Filter (per tap)	3.3 ns	2.5 ns	1.88 ns	1.1 ns
IIR Filter (per biquad)	13.3 ns	10 ns	7.5 ns	4.43 ns
On chip RAM	1 MB	2 MB	5 MB	5 MB

Performance for <$10



Performance for more $$

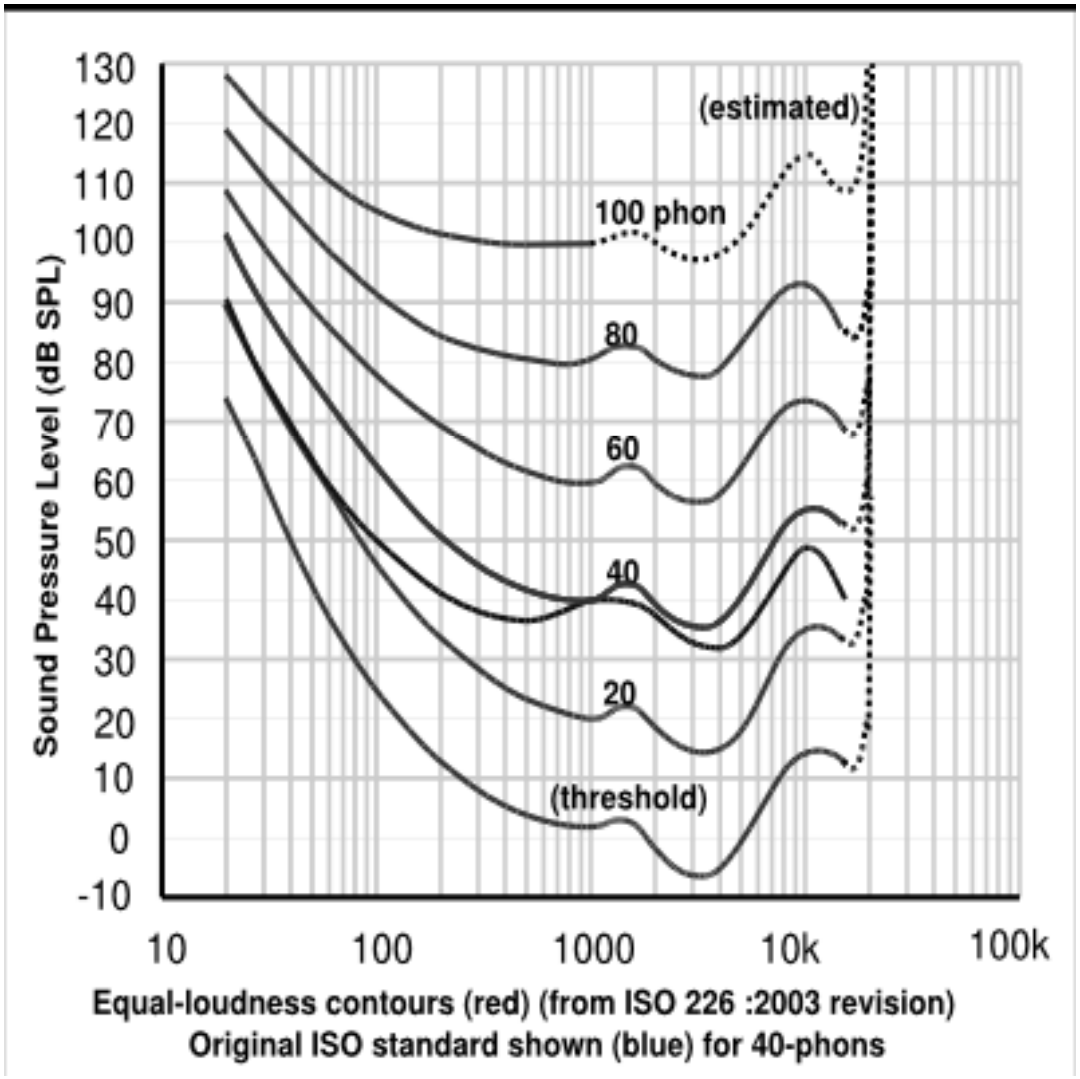


- **Latest BDTI numbers**
- **Next: Quick look at a DSP application**

Human Hearing

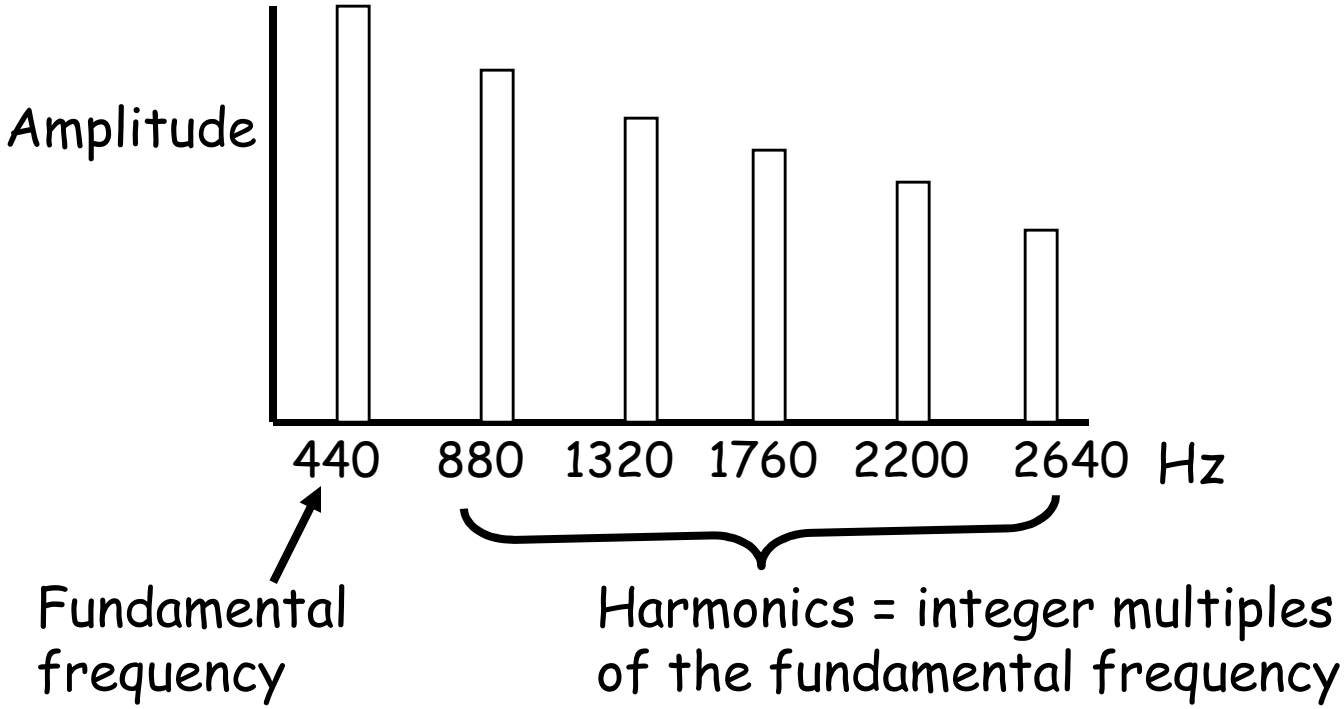
- The ear is basically a frequency spectrum analyzer
- Sound intensity measured in decibel sound power level
  - On a log scale
    - 20 dB = 10x change in air pressure
  - 0 dB = weakest detectable sound
  - 60 dB = normal speech
  - 140 dB = pain and damage
  - Ear can detect 1 dB change in volume
- Normal frequency range 20 Hz to 20 kHz
  - But most sensitive between 1 and 4 kHz

Equal Loudness Curves



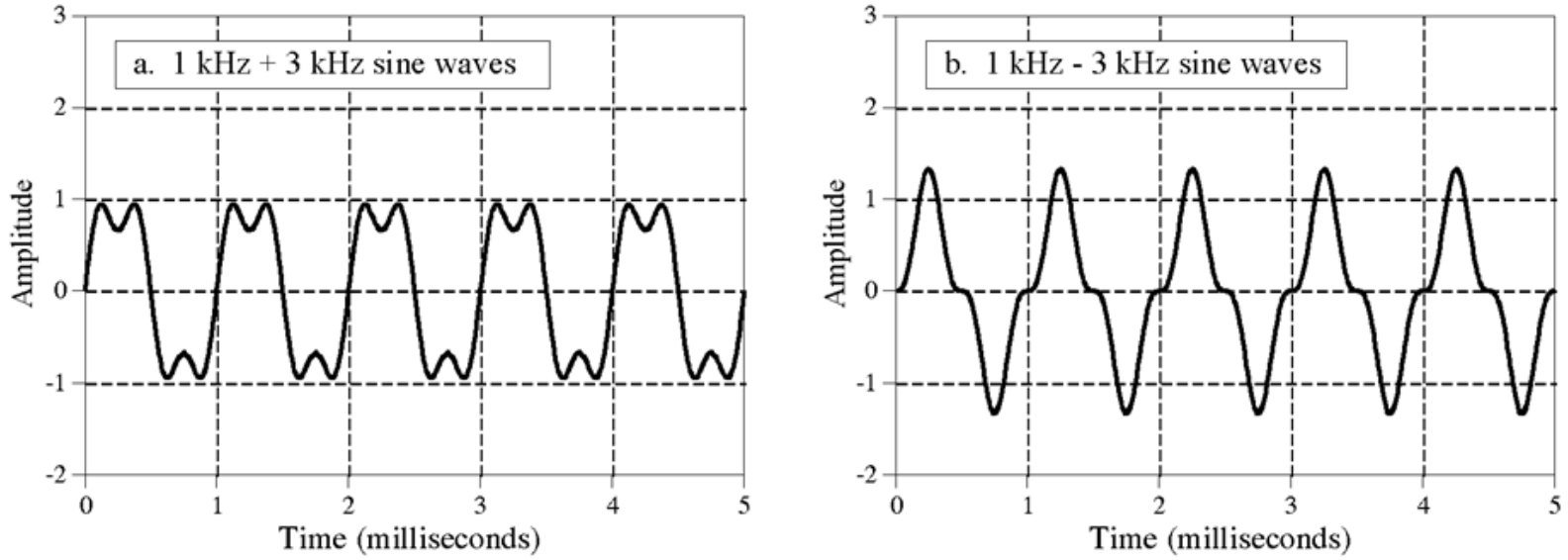
More Hearing

- **We perceive**
  - **Loudness**
  - **Pitch**
  - **Timbre – harmonic content**



Phase Insensitivity

- Hearing is quite phase insensitive
- These waveforms sound the same:



- Why don't we hear phase?

Sound Quality vs. Data Rate

Quality	Bandwidth	Sampling rate	Number of bits	Data rate
CD	5 Hz-20 kHz	44.1 kHz	16	706 kbps
Telephone	200 Hz-3.2 kHz	8 kHz	12	96 kbps
Telephone with companding	200 Hz-3.2 kHz	8 kHz	8	64 kbps
Compressed speech	200 Hz-3.2 kHz	8 kHz	12	4 kbps

Why Look at Hearing?

- **Understanding hearing supports efficient audio processing**
  - **Alternative to understanding is overkill**
  - **E.g., CD-quality audio**

- **MP3 exploits limitations of hearing**
  - **Notes with similar frequencies cannot be distinguished**
  - **Sounds close in time cannot be distinguished**
  - **Loud notes drown quieter ones**
  - **Ear is not uniformly sensitive to all frequencies**

MP3 Encoding

1. **Break data into frames**
2. **Convert into frequency domain**
3. **Use psychoacoustic model to sort frequency components by importance**
   - **Drop less important components subject to bit-rate constraints**
4. **Perform Huffman encoding on coefficients**
5. **Put frame data together into a bit stream**

- **Which of these are DSP-intensive?**

Summary

- **DSPs are cool**
  - **Far more bang for the buck than microcontrollers for signal processing**
  - **Interesting instruction sets, architectures, and compilers**
- **Sound processing**
  - **Significant user of DSP chips**
  - **Need to understand capabilities / limitations of human hearing**