Lesson 101: Audio compression

Can be used in a number of applications:

- Digital telephony
- Digital music
- Voice recording
- ...

Audio compression – A law, μ law

- With the same quantization step, the relative quantization error is larger for small signal levels than for large signal levels
- Idea: if we increase the quantization step for large signal levels, while we keep it as it is for small signals levels, the maximum relative quantization error remains the same
- Advantage: large quantization step means lower bits to represent the quantized signal, that is, we achieved compression
- Uniform Pulse Code Modulation (PCM) is an encoding method where the quantizer values are uniformly spaced.
- Logarithmic (A or μ) PCM allows 8 bits per sample to represent the same range of values that would be achieved with 14 bits per sample uniform PCM.
- This translates into a compression ratio of 1.75:1 (original amount of information:compressed amount of information).

Audio compression – A law, μ law (cont'd)

- North American μ-law quantizer (that is, PCM-to-μ-law):

$$y = \frac{\ln(1+\mu x)}{\ln(1+\mu)}$$

where $0 \leq x \leq 1$ and μ is a parameter ranging from 0 (no compression) to 255.

- European A-law quantizer (that is, PCM-to-A-law):

$$y = \frac{Ax}{1+\ln A} \quad \text{where} \quad 0 \leq x \leq \frac{1}{A}$$

$$y = \frac{1+\ln(Ax)}{1+\ln A} \quad \text{where} \quad \frac{1}{A} \leq x \leq 1$$

where A $=$ 87.6 and X is the normalized integer to be compressed.

- **How to implement the logarithm using integer arithmetic while achieving a low computing time?**

University of Victoria
British Columbia • Canada

Audio compression – how to implement the logarithm

- Since multiplications and divisions by 2 are simple shift operations: would it be better to implement $\log_2$ rather than $\ln$?
  - The answer is likely YES
- The difference between logarithms in different bases is only a factor of scale.

$$\log_N A = \log_N M \cdot \log_M A$$

- Brute force: build a *Look-Up Table* (LUT) with 14 inputs and 8 outputs that stores the logarithmic function – quite expensive in terms of silicon area
- To reduce the LUT size: divide the input interval into subintervals and provide a smaller LUT per subinterval – conceptually, the problem is only forwarded to subinterval level.

Audio compression – how to implement the logarithm

- **Taylor series expansion** about a point – approximation good for 1 point

$$\ln(1+x) = x - \frac{1}{2}x^2 + \frac{1}{3}x^3 - \frac{1}{4}x^4 + \ldots$$

(recall from Mathematics: this is Taylor series expansion about $x_0 = 0$)

- **Tchebishev polynomial** – approximation good for an interval (homework)

- **Piecewise linear approximation**
  - A particular case of series expansion
  - Easy to implement but precision may be an issue

University of Victoria
British Columbia • Canada

The logarithm: Taylor series expansion

- The formula (expansion about $x_0 = 0$):

$$\ln(1+x) = x - \frac{1}{2}x^2 + \frac{1}{3}x^3 - \frac{1}{4}x^4 + \ldots$$

- From the digital processor point of view, Taylor series expansion is an expensive approach in terms of the type of operations (multiplications, divisions), the number of operations, and the wordlength needed to achive the desired precision.

- An idea! Can we approximate the logarithm using only the first (linear) term?

$$\ln(1+x) \approx x \qquad \text{about } x_0 = 0$$

- It is possible if the precision is adequate for our task (and, fortunately, it is adequate according to the μ-law standard).

- Recall that the linear approximation is good only about $x_0 = 0$.

The logarithm: piecewise linear approximation

- Linear approximation:

$$\ln(1+x) \approx x \qquad \text{about } x_0 = 0$$

- To approximate over a large range of values, we have to expand in Taylor series about more points; that is, we have to consider multiple linear segments

- This is referred to as **piecewise linear approximation**:

$$\ln(1+x) \approx \ln(1+x_0) + \frac{x-x_0}{1+x_0} \qquad \text{about } x_0$$

The logarithm: piecewise linear approximation

- Example with four segments:

$$\ln(1+x) \approx x \qquad \text{about } x_0 = 0$$

$$\ln(1+x) \approx 1 + \frac{x-(e-1)}{e} \qquad \text{about } x_0 = e-1 \approx 1.72$$

$$\ln(1+x) \approx 2 + \frac{x-(e^2-1)}{e^2} \qquad \text{about } x_0 = e^2-1 \approx 7.39$$

$$\ln(1+x) \approx 3 + \frac{x-(e^3-1)}{e^3} \qquad \text{about } x_0 = e^3-1 \approx 19.09$$

$$\ldots$$

Assumptions for the sake of presentation

- Consider the μ-law quantizer:

$$y = \frac{\ln(1+\mu x)}{\ln(1+\mu)}$$

where $0 \le x \le 1$ and μ is a parameter ranging from 0 (no compression) to 255.

- For $\mu = 15$ we will approximate, in fact, $\log_2$

$$\frac{\ln(1+\mu x)}{\ln(1+\mu)} = \frac{\ln(1+15x)}{\ln(16)} = \frac{\ln(1+15x)}{4\ln(2)} = \frac{1}{4}\log_2(1+15x)$$

- Computing $\log_2$ is likely to be easier than $\ln$, since multiplications and divisions with 2 are simple shift operations.

- Homework: analyze $\mu \neq 15$

University of Victoria
British Columbia • Canada

Piecewise linear approximation of $\log_2()$

- Assume $\log_2()$:

$$\log_2(x) = \begin{cases} x-1 & \text{if } 1 \le x < 2, \\ x/2 & \text{if } 2 \le x < 2^2, \\ x/2^2+1 & \text{if } 2^2 \le x < 2^3, \\ x/2^3+2 & \text{if } 2^3 \le x < 2^4, \\ x/2^4+3 & \text{if } 2^4 \le x < 2^5, \\ \ldots \end{cases}$$

- Note: all divisions are by powers of 2
- Homework: piecewise linear approximation for $\ln()$
- What is the error of the piecewise linear approximation?

Piecewise linear approximation using integer arithmetic

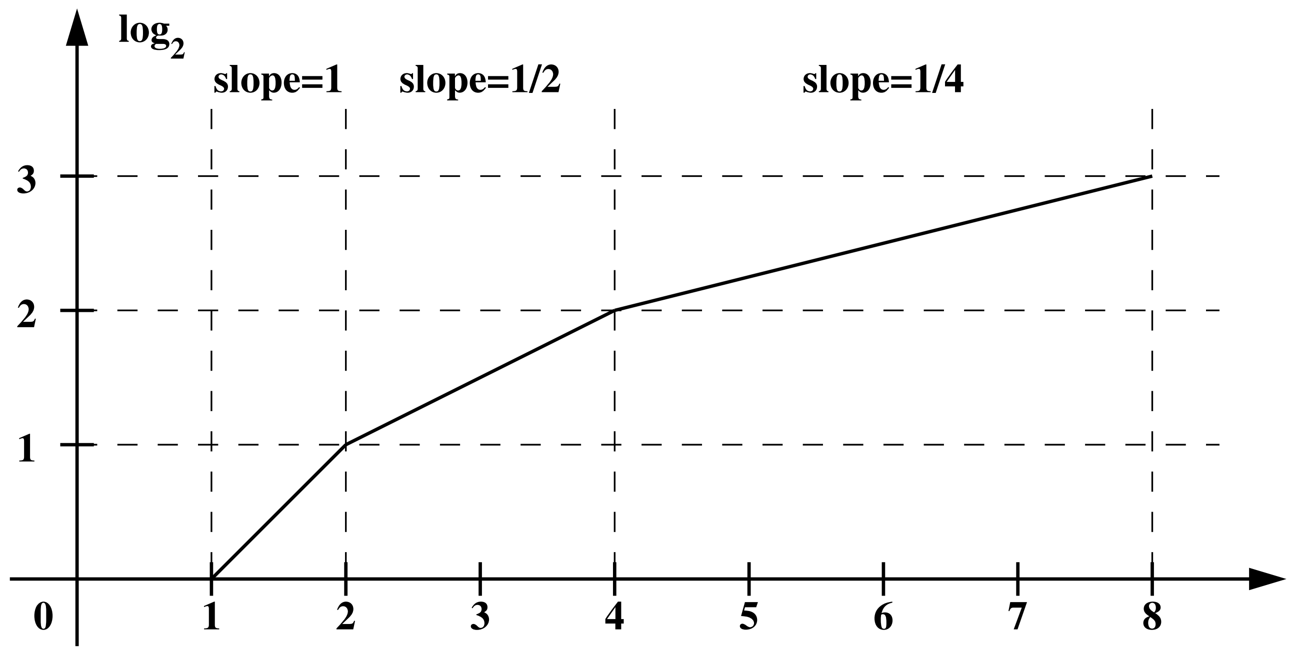
- Assume 12-bit unsigned integers and x ranging from 0 to 16:
  - 16 is represented as 2^{12}
  - 8 is represented as 2^{11}

    ...
  - 1 is represented as 2^8
  - x is represented as $X = 2^8\,x$
- Piecewise linear approximation for $\log_2$ using integer arithmetic:

$$\log_2(X) = \begin{cases} X - 2^8 & \text{if } 2^8 \le X < 2^9, \\ X/2 & \text{if } 2^9 \le X < 2^{10}, \\ X/2^2 + 2^8 & \text{if } 2^{10} \le X < 2^{11}, \\ X/2^3 + 2^9 & \text{if } 2^{11} \le X < 2^{12}. \end{cases}$$

- $\log_2(X)$ is an unsigned integer ranging $0 \ldots 2^{10}$

Piecewise linear approximation of $\log_2()$



$$\log_2(x) \approx \begin{cases} x - 1 & \text{if } 1 \leq x < 2, \\ 1 + \dfrac{x-2}{2} & \text{if } 2 \leq x < 4, \\ 2 + \dfrac{x-4}{4} & \text{if } 4 \leq x < 8, \\ \ldots \end{cases}$$

University of Victoria
British Columbia • Canada

C code for piecewise linear approximation of $\log_2()$

```
float pwlog2( unsigned char x) {
/* pwlog2 = piecewise log2 */
  if( x < 1) then
    return( -1); /* error */
  if( x < 2) then
    return( x-1);
  if( x < 4) then
    return( 1 + (x-2)/2);
  if( x < 8) then
    return( 2 + (x-4)/4);
  if( x < 16) then
    return( 3 + (x-8)/8);
  if( x < 32) then
    return( 4 + (x-16)/16);
  if( x < 64) then
    return( 5 + (x-32)/32);
  if( x < 128) then
    return( 6 + (x-64)/64);
  if( x < 256) then
    return( 7 + (x-128)/128);
}
```

- Simple code just for debugging purpose
  - The function returns a float value, which is not what we want
  - The way to compute the interpolation (1 + (x-2)/2) gives the compiler the full freedom to cast variables from integer type to float type.

C code for piecewise linear approximation of $\log_2()$ (cont'd)

- We want a function **pwlog2()** that returns an integer!
- Assume that argument x is an 8-bit unsigned integer, that is, it ranges from 0 to 256 (0 is never used, since $\log_2(0) = -\infty$
- $\log_2(x)$ is a real value ranging from 0 to 8
- Assume we want $\text{pwlog2}(x)$ to be represented also on an 8-bit unsigned integer
  - $x = 1$ corresponds to $32 \times \text{pwlog2}(1) = 0 = 00_{\text{h}}$
  - $x = 255$ corresponds to $32 \times \text{pwlog2}(255) = 32 \times 7.99219 = 255.75 = FF_{\text{h}}$
  - $x = 254$ corresponds to $32 \times \text{pwlog2}(254) = 32 \times 7.98437 = 255.50 = FF_{\text{h}}$
  - ... etc.
- We can rewrite the code in order to use only integer arithmetic

C code for piecewise linear approximation of $\log_2()$ (cont'd)

```
unsigned char pwlog2(
  unsigned char x) {
/* pwlog2 = piecewise log2 */
  if( x < 1)
    return( 0); /* error */
  if( x < 2)
    return((x-1) << 5);
  if( x < 4)
    return((1<<5) + ((x-2)<<4));
  if( x < 8)
    return((2<<5) + ((x-4)<<3));
  if( x < 16)
    return((3<<5) + ((x-8)<<2));
  if( x < 32)
    return((4<<5) + ((x-16)<<1));
  if( x < 64)
    return( (5<<5) + (x-32));
  if( x < 128)
    return( (6<<5) + ((x-64)>>1));
  if( x < 256)
    return( (7<<5) + ((x-128)>>2));
}
```

- Multiplication by 32 is left-shifting by 5
- Division by 2 is right-shifting, and can be done with or without rounding
  - We still have problems!

Division by a power of 2 (right-shifting)

- Let's first see two examples in base-10

Division by 10	Division by 100
$\frac{14584}{10} = 1458.4 \approx 1458$	$\frac{14584}{100} = 145.84 \approx 146$
$\frac{59117}{10} = 5911.7 \approx 5912$	$\frac{59117}{100} = 591.17 \approx 591$

- Rounding: if the fractional part is ≤ 0.5 then add 1, otherwise do nothing

University of Victoria
British Columbia • Canada

Division by a power of 2 (right-shifting)

- Division by 2

$$\frac{210}{2} = \frac{11010010_2}{2} = 1101001.0_2 \approx 1101001_2 = 105$$
$$\frac{211}{2} = \frac{11010011_2}{2} = 1101001.1_2 \approx 1101010_2 = 106$$

- Division by 4

$$\frac{210}{4} = \frac{11010010_2}{4} = 110100.10_2 \approx 110101_2 = 53$$
$$\frac{209}{2} = \frac{11010001_2}{4} = 110100.01_2 \approx 110100_2 = 52$$

University of Victoria
British Columbia • Canada

Division by a power of 2 (right-shifting)

- Rounding: if the fractional part is 0.1 then add 1, otherwise do nothing
- This is the same with:
  - Division by 2: `(x+1) >> 1`
  - Division by 4: `(x+2) >> 2`
  - Division by 8: `(x+4) >> 3`
  - Division by 16: `(x+8) >> 4`
  - ...
- This is only one way to do rounding, and there are many other ways to do it
- Which way to do rounding is beyond the course scope – for details, check a book on Computer Arithmetic

Using the logarithm approximation

```
#include <stdio.h>

unsigned char a, b;

inline unsigned char pwlog2( unsigned char x) {
  ...
}

int main( void) {
  float c;
  scanf( "a = %i\n", &a);
  b = pwlog2( a);
  c = b / 32.0;
  printf( "log2( a) = %f\n", c);
}
```

The assembly code for the logarithm approximation

```
pwlog2:
        mov     ip, sp
        stmfd   sp!,{fp,ip,lr,pc}
        sub     fp, ip, #4
        mov     r3, r0
        and     r2, r3, #255
        mov     r3, r2
        cmp     r3, #0
        bne     .L3
        mov     r0, #0
        b       .L2
.L3:
        cmp     r3, #1
        bhi     .L4
        mov     r2, r3
        sub     r1, r2, #1
        mov     r2, r1
        mov     r1, r2, asl #5
        and     r2, r1, #255
        mov     r0, r2
        b       .L2
.L4:
        cmp     r3, #3
        bhi     .L5
        mov     r2, r3
        sub     r1, r2, #2
        mov     r2, r1
        mov     r1, r2, asl #4
        mov     r2, r1
        add     r1, r2, #32
        and     r2, r1, #255
        mov     r0, r2
        b       .L2
```

University of Victoria
British Columbia • Canada

The assembly code for the logarithm approximation (cont'd)

```
.L5:
        cmp     r3, #7
        bhi     .L6
        mov     r2, r3
        sub     r1, r2, #4
        mov     r2, r1
        mov     r1, r2, asl #3
        mov     r2, r1
        add     r1, r2, #64
        and     r2, r1, #255
        mov     r0, r2
        b       .L2
```

```
L6:
        cmp     r3, #15
        bhi     .L7
        mov     r2, r3
        sub     r1, r2, #8
        mov     r2, r1
        mov     r1, r2, asl #2
        mov     r2, r1
        add     r1, r2, #96
        and     r2, r1, #255
        mov     r0, r2
        b       .L2
```

The assembly code for the logarithm approximation (cont'd)

```
.L7:
        cmp     r3, #31
        bhi     .L8
        mov     r2, r3
        sub     r1, r2, #16
        mov     r2, r1
        mov     r1, r2, asl #1
        mov     r2, r1
        sub     r1, r2, #128
        and     r2, r1, #255
        mov     r0, r2
        b       .L2
```

```
.L8:
        cmp     r3, #63
        bhi     .L9
        add     r1, r3, #128
        and     r2, r1, #255
        mov     r0, r2
        b       .L2
```

University of Victoria
British Columbia • Canada

The assembly code for the logarithm approximation (cont'd)

```
.L9:
        mov     r1, r3, asl #24
        mov     r2, r1, asr #24
        cmp     r2, #0
        blt     .L10
        sub     r2, r3, #64
        mov     r1, r2, asr #1
        mov     r2, r1
        sub     r1, r2, #64
        and     r2, r1, #255
        mov     r0, r2
        b       .L2
```

```
.L10:
        sub     r2, r3, #128
        mov     r1, r2, asr #2
        mov     r2, r1
        sub     r1, r2, #32
        and     r2, r1, #255
        mov     r0, r2
        b       .L2
.L11:
.L2:
        ldmea   fp, {fp, sp, pc}
```

The assembly code for the logarithm approximation (cont'd)

```
main:
        mov     ip, sp
        stmfd   sp!,{fp,ip,lr,pc}
        sub     fp, ip, #4
        sub     sp, sp, #8
        ldr     r0, .L14
        bl      printf
        sub     r3, fp, #16
        ldr     r0, .L14+4
        mov     r1, r3
        bl      scanf
        ldr     r3, .L14+8
        ldrb    r2, [fp, #-16]
        strb    r2, [r3, #0]
        ldr     r3, .L14+8
        ldrb    r2, [r3, #0]
        mov     r0, r2
        bl      pwlog2
        mov     r3, r0
        ldr     r2, .L14+12
        strb    r3, [r2, #0]
        ldr     r3, .L14+12
        ldrb    r2, [r3, #0]
        ldr     r0, .L14+16
        mov     r1, r2
        bl      printf
        ldr     r3, .L14+12
        ldrb    r2, [r3, #0]
        ...
        ldmfd   sp!, {r1, r2}
        bl      printf
        b       .L13
```


Performance estimation

- Estimating the overhead: 74327 cycles

```
volatile unsigned char a, b;

int main( void) {
  float c;
  scanf( "a = %i\n", &a);
  b = a;
  c = b / 32.0;
  printf( "log2( a) = %f\n", c);
}
```

Performance estimation (cont'd)

- Use the trick: execute a large number of times the program core: 1255100 cycles

```
volatile unsigned char a, b;

int main( void) {
  int l;
  float c;
  scanf( "a = %i\n", &a);
  for( l=0; l<65536; l++)
    b = a;
  c = b / 32.0;
  printf( "log2( a) = %f\n", c);
}
```

Performance estimation (cont'd)

- Estimating the performance of `pwlog2()`: 3809811 cycles (a = 3)

```
volatile unsigned char a, b;

int main( void) {
  int l;
  float c;
  scanf( "a = %i\n", &a);
  for( l=0; l<65536; l++)
    b = pwlog2( a);
  c = b / 32.0;
  printf( "log2( a) = %f\n", c);
}
```

Performance estimation (cont'd)

- Assume we run `pwlog2()` 65536 times in a loop
- For a = 3, `pwlog2()` takes: (3809811 - 1255100) / 65536 = 39 cycles
- For a = 250, `pwlog2()` takes: (4859009 - 1255100) / 65536 = 55 cycles
- The performance is data dependent
- Homework: assuming a uniform distribution of the input data, can we do better?

Architectural support for pwlog2()

- Numerical figures for the entire program:
  - For simple assignment: 1255100 / 65536 = 19 cycles
  - With simple function call: 2893298 / 65536 = 44 cycles
  - With `pwlog2()` function call: 3809811 / 65536 = 58 cycles
- Significant overhead for function call!
- Would be great to call `pwlog2()` without overhead
- Idea: define a new instruction `PWLOG2` (extend the instruction set architecture), and build the corresponding computing unit that calculates `pwlog2()`
- The **new instruction** needs to be instantiated
  - Augment the compiler – beyond the course scope
  - Use intrinsics (custom operations)
- Build a **custom computing unit**
  - Design full-custom hardware (zeroth-, first-, second-order systems)
  - Microcode solutions (third-order system)

University of Victoria
British Columbia • Canada

Architectural support for pwlog2() (cont'd)

```
volatile a, b;

int main( void) {
  int d;
  scanf( "%d", &a);
  __asm__ ( "PWLOG2 %1, %2, %0" : "=r" (b) : "r" (a), "r" (dummy));
  printf( "log2( a) = %i\n", b);
}
```

- Generate the assembly file

The assembly code with the new PWLOG2 instruction

```
main:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        sub     sp, sp, #16
        ldr     r0, .L5
        bl      printf
        sub     r3, fp, #16
        ldr     r0, .L5+4
        mov     r1, r3
        bl      scanf
        ldr     r3, .L5+8
        ldrb    r2, [fp, #-16]
        strb    r2, [r3, #0]
        ldr     r3, .L5+8
        ldrb    r2, [r3, #0]
        ldr     r3, [fp, #-24]
        PWLOG2 r2, r3, r2
        ldr     r3, .L5+12
        strb    r2, [r3, #0]
        ldrb    r3, [r3, #0]
        ldr     r2, .L5+12
        ldrb    r1, [r2, #0]
        and     r3, r1, #255
        ldr     r0, .L5+16
        mov     r1, r3
        bl      printf
        ...
```

Instead of a function call, we have an instruction!

University of Victoria
British Columbia • Canada

Extending the ARM instruction-set architecture

- Although we tightly optimized our function, we may find that it is now fast enough, and the overhead to call this function is still high
- Next step: implement the function in firmware/hardware and define a new instruction to call the new computing unit
- This process is referred to as *extending the instruction-set architecture*
- Since a single instruction replaces a function call, the overhead associated to function call does not exist any longer
- The new computing unit should be tightly optimized
  - The microcode engine can be either vertical and horizontal, with a specific instruction set
  - The custom hardware includes an automaton whose structure is geared to perform our function

Audio compression – project requirements

- Use the piecewise linear approximation for $\ln(x)$
- Determine the maximum error when using piecewise linear approximation
- Implement piecewise linear approximation using integer arithmetic for $\ln(X)$ in
  - software (write C routines)
  - custom hardware (write VHDL/Verilog)
- Compress an audio string using piecewise linear approximation of $\ln$ and estimate:
  - the performance improvement of hardware-based solution versus software-based solution
  - the performance improvement of a 2-issue slot firmware-based solution versus software-based solution

Questions, feedbacks

?

University of Victoria
British Columbia • Canada