Lesson 1: Audio compression

University of Victoria
British Columbia • Canada

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

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

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

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

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

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 word-width 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 $0 \leq x \leq 1$ 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 $\mu \neq 15$

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?

University of Victoria
British Columbia • Canada

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 $X = 2^8\,x$

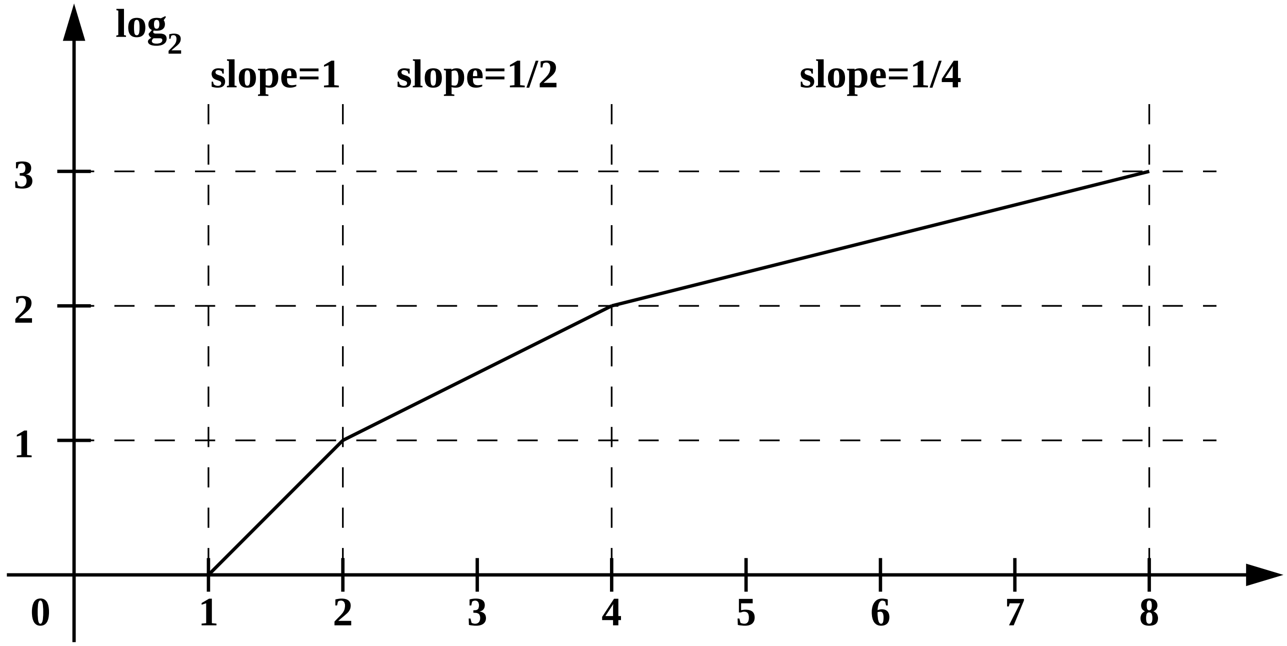
- Piecewise linear approximation for $\log_2$ using integer arithmetic:

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

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

University of Victoria
British Columbia • Canada

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 $\mathrm{pwlog2}(x)$ to be represented also on an 8-bit unsigned integer
  - $x = 1$ corresponds to $32 \times \mathrm{pwlog2}(1) = 0 = 00_\mathrm{h}$
  - $x = 255$ corresponds to $32 \times \mathrm{pwlog2}(255) = 32 \times 7.99219 = 255.75 = FF_\mathrm{h}$
  - $x = 254$ corresponds to $32 \times \mathrm{pwlog2}(254) = 32 \times 7.98437 = 255.50 = FF_\mathrm{h}$
  - ... 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

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

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
```

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
```

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);
}
```

University of Victoria
British Columbia • Canada

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)

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

Lesson 2: Huffman Encoding and Decoding

Huffman (variable-length) coding

- Optimal encoding with respect to transmission rate
- Based on the probability of each symbol
  - Uses a variable-length code table for encoding a source symbol
  - The code-length depends on the probability of occurrence
- Let us assume a 5-symbol alphabet having the following probability distribution:
  A / 0.4, **B** / 0.3, **C** / 0.15, **D** / 0.1, **E** / 0.05
- Encode in a way that minimizes the transmission rate:
  - **A** – 0
  - All the others – 1
    - **B** – 0, that is **B** is 10
    - All the others – 1
      - **C** – 0, that is **C** is 110

- All the others – 1
- ...

Hufmann encoding

- The coding table:

Symbol	Bit combination	Code-length
A	0	1
B	10	2
C	110	3
D	1110	4
E	1111	4

- 3 bits are needed to represent the alphabet symbols
  - Transmission rate: 3 bits/cycle
- Between 1 and 4 bits are needed to represent the code-words
  - Transmission rate: 2 bits/cycle
    $(0.4 \times 1 + 0.3 \times 2 + 0.15 \times 3 + 0.1 \times 4 + 0.05 \times 4 \approx 2)$

- Penalty: sequential (slow) decoding process

Hufmann encoding

- Coding algorithm can rely on a reasonable small Look-Up Table (LUT)
  - For a 5-symbol alphabet: 3-input LUT with 4 outputs
    * This is a 32-bit memory
  - For a 128-symbol alphabet: 7-input LUT with 127 outputs
    * This is a 2KB memory
- A memory of 2KB should not be a problem even for an embedded system
- If the coding LUT is still too large for the considered embedded system
  - Subdivide the coding LUT into smaller LUTs and perform the coding process in several steps
  - Penalty: larger coding time
- What would a Huffman encoder implementation look like?
  - Huffman encoding does not pose difficult technical problems
  - Huffman decoding is a far more difficult task!

Possible Huffman encoder implementation strategies

- A single large LUT
  - The main code just access the LUT in order to retrieve the codeword
  - The LUT's word-width is equal to the longest codeword
- Several smaller LUTs
  - The LUT's word-width is smaller
  - The coding process is performed in several steps
- These strategies can be implemented both in:
  - Hardware: the LUT(s) are implemented within the functional unit
  - Software: the LUT(s) are stored into memory (ideally in cache)

Pure-software implementation of the Huffman encoder

```
#include <stdio.h>
char *HE_LUT[5] = { "0", "10", "110", "1110", "1111"};

int main( void) {
  char symbol_to_encode = 0;

  do {
    scanf( "%i", &symbol_to_encode);
    printf( "%s\n", HE_LUT[symbol_to_encode - 0x40]);
  } while ( (symbol_to_encode > 0x40) & (symbol_to_encode < 0x46));
  printf( "%s\n", "Not a valid symbol.");
  exit( 0);
}
```

- ASCII code of character 'A' is 0x41
- ASCII code of character 'E' is 0x45

Hufmann decoding

- A Hufmann-encoded string: `1101001110111101 0`

```
110  10  0  1110  1111  0  10
  C   B  A     D     E  A   B
```

- To achieve maximum compression, the coded data does not contain specific guard bits separating consecutive codewords
- The decoding process must:
  - Determine the symbol itself
  - Determine the code-length of the symbol
  - Shift the incoming string in order to discard the decoded bits
- Before initiating a new decoding iteration, the input string has to be shifted by a number of bits equal to the decoded code-length

  - A new symbol cannot be decoded before the current one has been decoded

- There are a lot of recursive operations that generate true-dependencies

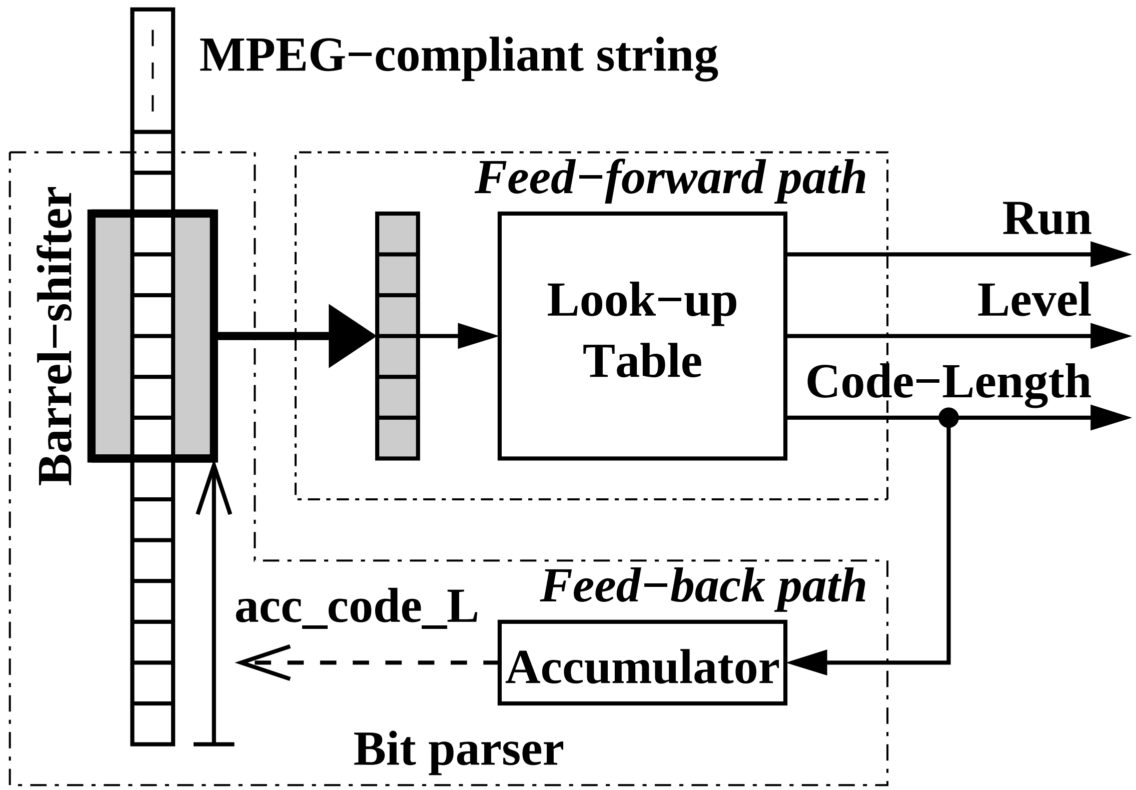
Hufmann decoding

- Hufmann decoding is intrinsically a sequential process
- Parallel processing capabilities are not likely to improve the decoding rate
  - Pipelined engine
  - Horizontal engine
- Providing Huffman decoding hardware support is worth to be considered
- Will the processor be idle while the Huffman unit decodes the input string?
- Combine Huffman decoding with other tasks, for example:
  - Run-Length Decoding (RLD)
  - Inverse Discrete Cosine Transform (IDCT)

Hufmann decoding – the brute force approach

- Select a chunck of the incoming string that has a number of bits equal to the largest code-length
- Look-up into a Huffman decoding table with the selected chunck as address
- The LUT returns:
  - The bit combination of the decoded symbol
  - The code-length of the decoded symbol
- Discard *code-length* bits from the incoming string
- This approach is good for very small code-lengths since the LUT is small
- For large code-lengths the LUT size becomes very large!
  - MPEG: the longest codeword (excluding Escape!) is 17 bits $\longrightarrow$ the LUT size reaches $2^{17} = 128$ K words for a direct mapping of all possible codewords
  - MPEG: the symbol is a combination of a *run* code and a *length* code

Huffman (variable-length) decoding principle



- VLD performance: the throughput is bounded by the inverse to the loop latency

Huffman (variable-length) decoding principle

- VLD is a system with feedback, whose loop typically contains:
  - Look-Up Table on the feed-forward path
  - Bit parser on the feedback path

- LUT receives the variable-length code itself as an address and outputs:
  - the decoded symbol (*run-level* pair or *end_of_block*)
  - the codeword length

- To determine the starting position of the next codeword, the *code_length* is fed back to an accumulator and added to the previous sum of codeword lengths,

- The bit parsing operation is completed by the *barrel-shifter* (or *funnel-shifter*) which shifts out the decoded bits.

Huffman (variable-length) decoding performance

- The throughput is bounded by the inverse of the loop latency
- Major goal: reduce the loop latency!
  - Reduce the operation budget
    - Look-up operation
    - Accumulation
    - Barrel-shifting
  - Reduce the latency of each operation
- Hardware issues regarding VLD parts
  - Barrel-shifter is essentially a DEMUX – implemented within the standard instruction set (that is, in software)
  - Adder that performs the accumulation should be high-performance (carry look-ahead, carry select, etc.)
  - LUT: low latency is more important than silicon area

Huffman decoding: reducing the operation budget

- Keep the accumulator out of the critical path:

  M.-T. Sun, *VLSI architecture and Implementation of a High-Speed Entropy Decoder*, Proceedings of the IEEE International Symposium on Circuits and Systems, 1991, pp. 200-203.

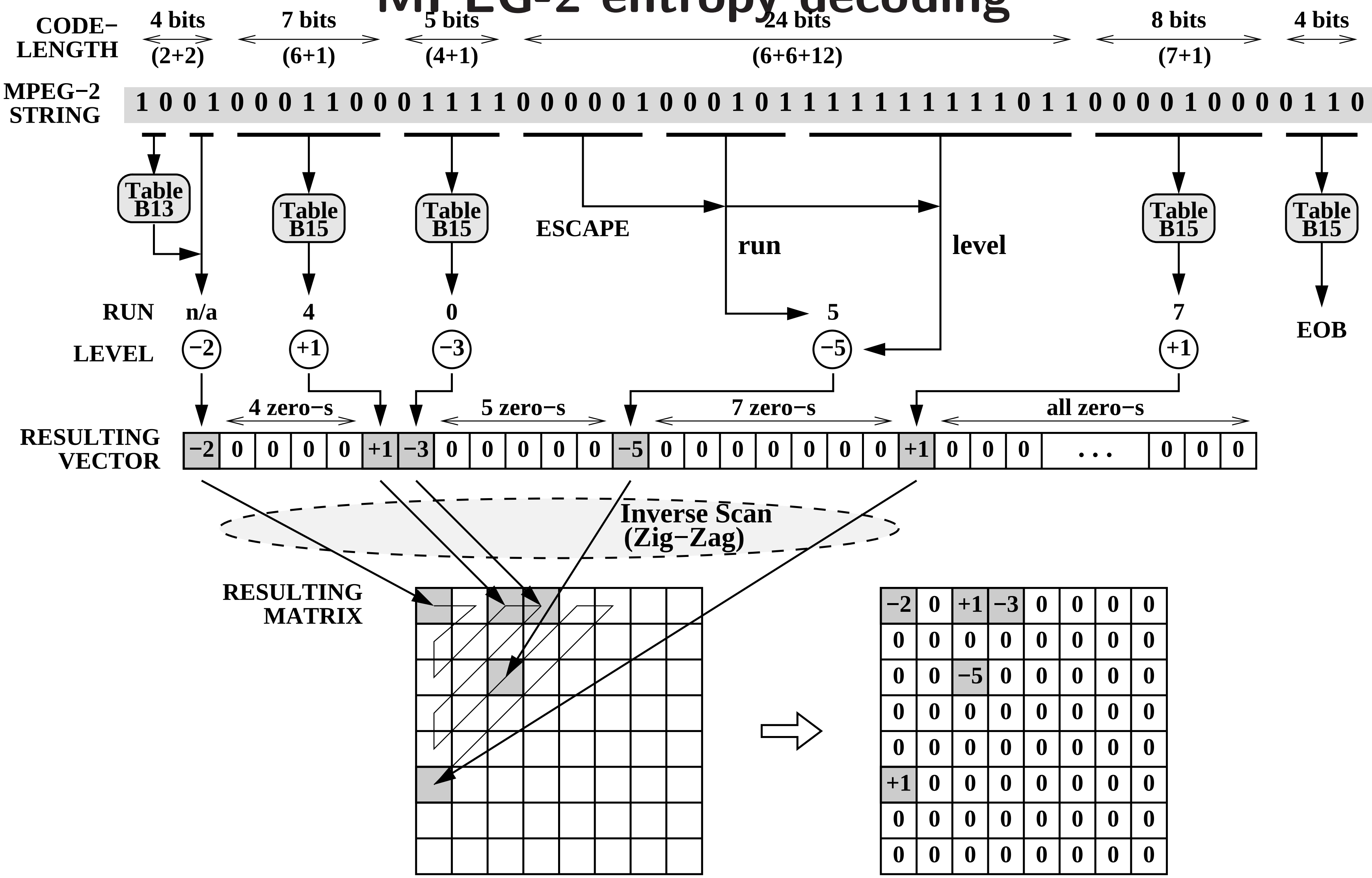
- Is multiple-symbol decoding possible?
  - What is really important is to detect the code-lengths to be able to initiate the next decoding iteration
  - What would be the LUT size in this case? Try multiple-symbol decoding for short codewords and single symbol decoding for long codewords.

- Try to split the accumulation operation is plain addition and storage

MPEG: Entropy decoding

- MPEG video coding standard:
  - DCT + Quantization: lossy compression
  - Entropy coding: lossless compression
- Entropy decoding consists of two distinct steps:
  - Variable-Length (Huffman) Decoding (VLD)
  - Run-Length Decoding (RLD)
- Both VLD and RLD are sequential tasks (due to data dependencies)
- Entropy decoding is an intricate function on parallel computing engines
- Entropy decoding is an ideal candidate to benefit from hardware support.

MPEG-2 entropy decoding



−2	0	+1	−3	0	0	0	0
0	0	0	0	0	0	0	0
0	0	−5	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
+1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Hufmann decoding – project requirements

- Define your own alphabet
- Assume a particular distribution for the probabilities of occurence
- Define the Huffman codes and calculate the average transmission rate with and without Huffman coding
- Build the testbench (= a file that contains alphabet symbols occuring with the assumed probabilities)
- Provide a pure-software solution for Huffman decoding
  - Try to reduce the cache misses (do not use very large LUTs)
  - Estimate the performance for the particular testbench
- Try also a firmware solution, but since Huffman decoding is a sequential process do not expect any improvement

Hufmann decoding – project requirements

- Build a full-custom hardware unit for the Huffman decoder and estimate its performance against 32-bit addition
  - Reentrant or non-reentrant functional unit?
- Define a new instruction that will call the full-custom Huffman decoder
  - You must comply with the ARM architecture (you can have at most two arguments and one result per instruction call)
- Rewrite the high-level code and instantiate the new instruction
  - Use assembly inlining
- Estimate the performance of the ARM processor augmented with a Huffman decoding unit
- Estimate the speed-up (if any) and the penalty in terms of number of gates required to implement the Huffman decoder

Lesson 3: Matrix inversion

- Condition number – recall from Linear Algebra
- Matrix inversion by cofactor expansion
- Matrix inversion by Gaussian elimination
- Matrix inversion – project requirements

Condition number – definition

- Goal: to measure how the error in the input data affects the computed answer
- Example for computing the derivative:

$$f'(x) = \frac{f(x+\delta x) - f(x)}{\delta x} \quad \Longrightarrow \quad \frac{|f(x+\delta x) - f(x)|}{|f(x)|} = \frac{|\delta x|}{|x|} \times \frac{|f'(x)| \cdot |x|}{|f(x)|}$$

  where:
  - $|\delta x|/|x|$ is the relative error in the input
  - $|f(x+\delta x) - f(x)|/|f(x)|$ is the relative error in the output
  - $|f'(x)| \cdot |x|/|f(x)|$ is the **condition number**

- For each problem there is a condition number that we have to derive
  - Finding the derivative has a condition number
  - Matrix inversion has a condition number, $\kappa = ||A|| \cdot ||A^{-1}||$
  - Finding the eigenvalues has a different condition number

Condition number for matrix inversion, κ

- Matrix inversion is typically analized in connection with a system:

$$\mathbf{A}x = b \quad \Longrightarrow \quad x = \mathbf{A}^{-1}b$$

- The condition number, $\kappa = ||A|| \cdot ||A^{-1}||$, is the relative change $||\delta x||/||x||$ in the answer as a multiple of the relative change $||\delta A||/||A||$ in the data
- We simply multiply the condition number by a bound on the input error to get a bound on the computed solution
- Goal: to estimate the condition number without calculating the norm explicitly
- We need a way to approximate the matrix norm
  - There are many ways for that

Condition number for matrix inversion, κ

- Approximating the **matrix norm**: the maximum absolute row sum

$$||A|| = \max_i \sum_j |a_{ij}|$$

- Condition number is kind of a "magnification factor", $\kappa = ||A^{-1}|| \cdot ||A||$

- **A is well conditioned**

  κ is small relative to 1 $\longrightarrow$ a small relative change (or error) in A cannot produce a large relative change (or error) in the inverse

- **A is ill conditioned**

  κ is large $\longrightarrow$ a small relative change (or error) in A can possibly (but not necessarily) result in a large relative change (or error) in the inverse

University of Victoria
British Columbia • Canada

Condition number for matrix inversion, κ

- We have seen that the following system is sensitive to small perturbations

$$835x + 667y = 168$$
$$333x + 266y = 67$$

- The system matrix, A:

$$\mathbf{A} = \begin{pmatrix} 835 & 667 \\ 333 & 266 \end{pmatrix}$$

- The matrix A is not singular, its inverse does exist:

$$\mathbf{A}^{-1} = \begin{pmatrix} -266 & 667 \\ 333 & -835 \end{pmatrix}$$

- Obviously, $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$

Condition number for matrix inversion, κ

- Norm of $\mathbf{A}$:
  - $|835| + |667| = 1502$
  - $|333| + |266| = 599$
  - $||\mathbf{A}|| = \max\{1502, 599\} = 1502$
- Norm of $\mathbf{A}^{-1}$:
  - $|-266| + |667| = 933$
  - $|333| + |-835| = 1168$
  - $||\mathbf{A}^{-1}|| = \max\{933, 1168\} = 1168$
- Condition number for A:

$$\kappa = ||\mathbf{A}|| \cdot ||\mathbf{A}^{-1}|| = 1502 \cdot 1168 = 1754336 \approx 1.7 \cdot 10^6$$

- A relative change (or error) in the solution can be about a million times larger than the relative change (or error) in A

Matrix inversion by cofactor expansion

- Determination of a matrix that when multiplied by the given matrix will yield a unit matrix
- In terms of linear algebra, given a square n-by-n matrix A, find a square n-by-n matrix B (if one exists) such that AB = BA = In, the n-by-n identity matrix
- Brute force approach: calculate a matrix of cofactors:

$$A^{-1} = \frac{1}{|A|}(C_{ij})^T = \frac{1}{|A|}\begin{pmatrix} C_{11} & C_{21} & \dots & C_{j1} \\ C_{12} & \ddots & \vdots & C_{21} \\ \vdots & \dots & \ddots & \vdots \\ C_{1i} & \dots & \dots & C_{ji} \end{pmatrix}$$

  where $|A|$ is the determinant of A, C_{ji} is the matrix cofactor, and A^T represents the matrix transpose
- If $|A| = 0$ the inverse matrix does not exist

Matrix inversion by cofactor expansion (cont'd)

- The cofactor C_{ij} of A is defined as $(-1)^{i+j}$ times the minor M_{ij} of A
- The minor M_{ij} of A is the determinant of the smaller matrix that results from A by removing the i-th row and j-th column

$$A = \begin{pmatrix} 1 & 4 & 7 \\ 3 & 0 & 5 \\ -1 & 9 & 11 \end{pmatrix} \qquad C_{23} = (-1)^{2+3} \begin{vmatrix} 1 & 4 \\ -1 & 9 \end{vmatrix} = (-1)(9+4) = -13$$

- The determinant
  - If A is a 1-by-1 matrix, then $|A| = A_{1,1}$
  - If A is a 2-by-2 matrix, then $|A| = A_{1,1}A_{2,2} - A_{2,1}A_{1,2}$
  - If A is a 3-by-3 matrix, then $|A| = A_{1,1}A_{2,2}A_{3,3} + A_{1,3}A_{3,2}A_{2,1} + A_{1,2}A_{2,3}A_{3,1} - A_{3,1}A_{2,2}A_{1,3} - A_{1,1}A_{2,3}A_{3,2} - A_{1,2}A_{2,1}A_{3,3}$

Matrix inversion by cofactor expansion (cont'd)

- For larger matrices, Laplace's formula can be used to expand a determinant along a row or column:

$$|A| = \sum_{j=1}^{n} A_{i,j} C_{i,j}$$

  where n is the number of elements in a row (column)

- Since this method is essentially recursive, it becomes inefficient for large matrices
- Determinant is a computationally-intensive task – it requires for multi-operand multiplication
- Hardware/firmware support for multi-operand multiplication is needed
- Even with hardware support for multi-operand multiplication the number of operations is huge ($> n!$)

Cofactor expansion is computationally inefficient

- Calculating the determinant of a matrix is a recursive process and requires a huge number of operations
  - The algorithm can start by choosing any one row or column
  - The determinant is the sum of the products of this row or column's values and sub-determinants formed by blocking out the row and column of the particular value
- The row or column can be choosen wisely to reduce the number of operations
  - If a certain row or column contains a few zeros, choosing it as the row/column that we take the determinant with respect to reduces the number of determinants to be calculated
- Better techniques to calculate the inverse are based on matrix factorization
  - Gaussian elimination and LU decomposition

– Gauss-Jordan elimination
– QR decomposition

Problems in matrix inversion

- Choose a matrix inversion algorithm with a not so large operation count
  - Matrix factorization (e.g., QR decomposition) is a good candidate
  - Cofactor expansion should typically be avoided
- Implement the matrix inversion algorithm in a way that exposes the parallelism (if any) to the compiler/processor
- Conditioning of the matrix
  - Adapt the inversion algorithm to the matrix (e.g., choose the appropriate number representation and the precision)
  - Redesign the task to get a well conditioned matrix
- Stability of the inversion algorithm
  - Gauss-Jordan elimination can be unstable without pivoting (row exchange)

  - QR decomposition is essentially stable without pivoting

- Since matrices are being manipulated, keep an eye on cache misses

Matrix inversion by Gauss-Jordan elimination

- The idea: convert a given system $\mathbf{A}x = b$ to an equivalent diagonal system by taking the appropriate linear combinations of the equations
  - Goal: vanish all matrix elements but one per column
- Example:

$$3x + 5y = 8$$
$$6x + 7y = 4$$

- Multiply the first equation by 2 and subtract it from the second:

$$3x + 5y = 8$$
$$-3y = -12$$

- Multiply the second equation by $5/3$ and add it to the first:

$$\begin{aligned} 3x \quad &= -12 \\ -\,3y &= -12 \end{aligned}$$

University of Victoria
British Columbia • Canada

Matrix inversion by Gauss-Jordan elimination

- Normalize the coefficients

$$
\begin{aligned}
x &= -4 \\
y &= 4
\end{aligned}
$$

- Same computation in matrix notation with the right-hand side:

$$
\begin{pmatrix} 3 & 5 & | & 1 & 0 \\ 6 & 7 & | & 0 & 1 \end{pmatrix} \longrightarrow \begin{pmatrix} 3 & 5 & | & 1 & 0 \\ 0 & -3 & | & -2 & 1 \end{pmatrix} \longrightarrow
$$

$$
\longrightarrow \begin{pmatrix} 3 & 0 & | & -7/3 & 5/3 \\ 0 & -3 & | & -2 & 1 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 0 & | & -7/6 & 5/6 \\ 0 & 1 & | & 2/3 & -1/3 \end{pmatrix}
$$

- The matrix inverse is:

$$
\begin{pmatrix} -7/6 & 5/6 \\ 2/3 & -1/3 \end{pmatrix}
$$

Gauss-Jordan elimination – Weaknesses

- All the right-hand sides to be stored and manipulated at the same time
  - Double memory footprint may induce cache misses
- When the inverse matrix is not desired, Gauss-Jordan is three times slower than the best alternative technique for solving a single linear set
- Gaussian elimination is not a stable algorithm
  - It can fail entirely, because it attempts division by zero
- Gaussian elimination with row interchanges (also known as Gaussian elimination with pivoting) is a better choice
  - Row interchanges may increase the traffic to and from memory

Instability of the Gaussian elimination without pivoting

- For certain matrices, it fails entirely, because it attempts division by zero
- Assuming the values are represented on 16-bit signed integers
  - The range: $-2^{15} \dots 2^{15} - 1 = -32,768 \cdots + 32,767$

$$\mathbf{A} = \begin{pmatrix} 0 & 2^{14} \\ 2^{14} & 2^{14} \end{pmatrix} = \begin{pmatrix} 0 & 16,384 \\ 16,384 & 16,384 \end{pmatrix}$$

- The matrix has full rank: $det(A) = -2^{28} \neq 0$
- The inverse matrix:

$$\mathbf{A}^{-1} = \begin{pmatrix} -2^{-14} & 2^{-14} \\ 2^{-14} & 0 \end{pmatrix}$$

- The matrix is well-conditioned: $\kappa(A) = 2^{15} \cdot 2^{-13} = 4$

- The Gaussian elimination fails at the first step $\longrightarrow$ pivoting required

Instability of the Gaussian elimination without pivoting

- The process does not fail if:

$$\mathbf{A} = \begin{pmatrix} 1 & 2^{14} \\ 2^{14} & 2^{14} \end{pmatrix} = \begin{pmatrix} 1 & 16,384 \\ 16,384 & 16,384 \end{pmatrix}$$

- Inverting the matrix:

$$\begin{pmatrix} 1 & 2^{14} & | & 1 & 0 \\ 2^{14} & 2^{14} & | & 0 & 1 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 2^{14} & | & 1 & 0 \\ 0 & -2^{14}(2^{14}-1) & | & -2^{14} & 1 \end{pmatrix} \longrightarrow$$

$$\longrightarrow \begin{pmatrix} 1 & 0 & | & 1-2^{14}/(2^{14}-1) & 1/(2^{14}-1) \\ 0 & -2^{14}(2^{14}-1) & | & -2^{14} & 1 \end{pmatrix} \longrightarrow$$

$$\longrightarrow \begin{pmatrix} 1 & 0 & | & 1-2^{14}/(2^{14}-1) & 1/(2^{14}-1) \\ 0 & 1 & | & 1/(2^{14}-1) & -1/[2^{14}(2^{14}-1)] \end{pmatrix}$$

Instability of the Gaussian elimination without pivoting

- The inverse is:

$$\mathbf{A}^{-1} = \begin{pmatrix} -1/(2^{14}-1) & 1/(2^{14}-1) \\ 1/(2^{14}-1) & -1/[2^{14}(2^{14}-1)] \end{pmatrix}$$

- Check the solution:

$$\begin{pmatrix} 1 & 2^{14} \\ 2^{14} & 2^{14} \end{pmatrix} \cdot \begin{pmatrix} -1/(2^{14}-1) & 1/(2^{14}-1) \\ 1/(2^{14}-1) & -1/[2^{14}(2^{14}-1)] \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

- The bottom right-hand element, $-1/[2^{14}(2^{14}-1)]$, is too small to be represented with 16 bits of precision

Instability of the Gaussian elimination without pivoting

- Let us scale up the inverse with 2^{14}:

$$2^{14}\mathbf{A}^{-1} = \begin{pmatrix} -2^{14}/(2^{14}-1) & 2^{14}/(2^{14}-1) \\ 2^{14}/(2^{14}-1) & -1/(2^{14}-1) \end{pmatrix} \approx \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix}$$

- Check the approximate solution:

$$\begin{pmatrix} 1 & 2^{14} \\ 2^{14} & 2^{14} \end{pmatrix} \cdot \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 2^{14}-1 & 1 \\ 0 & 2^{14} \end{pmatrix}$$

- This is called **numerical instability**

- **Homework**: reverse the order of the rows before proceeding

Gaussian elimination with pivoting

- Pivoting is also known as row interchange
- To avoid the instability of the standard Gaussian elimination, the largest element along the column is declared the pivot
- The matrix is stored into cache row-wise
- Finding the largest element along a column requires accessing the matrix column-wise, and that may generate cache misses
- **Pivoting does not come without a price**

Matrix inversion – project requirements

- Build a testbench containing:
  - A well-conditioned matrix
  - An ill-conditioned matrix
- Calculate the condition number and calculate the required precision dynamically
  - Make sure your matrix is not very ill-conditioned, otherwise more than 32 bits of precision will be required
- Implement Gauss-Jordan algorithm with pivoting using integer arithmetic
- Provide a pure-software solution and estimate its performance
- Provide hardware support for vanishing the column elements
  - Define a new instruction that drives the new hardware unit
  - Rewrite the code in order to instantiate the new instruction
- Compare the hardware-assisted solution with the software solution

Lesson 4: Matrix diagonalization

Singular Value Decomposition

- The Singular Value Decomposition (SVD) of a $n \times n$ matrix M is given by:

$$M = U \Sigma V^T$$

- U and V are orthogonal matrices and Σ is a diagonal matrix of singular values
- There are many methods to calculate SVD, Jacobi method is one of them
- The **Jacobi method** seeks to systematically reduce the off-diagonal elements to zero. This is done by applying a sequence of plane rotations to M which transforms M into Σ.
- Several sweeps over the entire matrix M may be necessary to complete the SVD.
- Within each sweep, the matrix elements need to be paired and appropriate rotations needs to be calculated. The $n \times n$ matrix is partitioned in $n/2 \times n/2$ blocks, each block being a 2×2 matrix.

Singular Value Decomposition – Jacobi method

- Assume the following matrix M:

$$M = \begin{pmatrix} m_{00} & \dots & m_{0i} & \dots & m_{0j} & \dots & m_{0n} \\ \vdots & & \vdots & & \vdots & & \vdots \\ m_{i0} & \dots & m_{ii} & \dots & m_{ij} & \dots & m_{in} \\ \vdots & & \vdots & & \vdots & & \vdots \\ m_{j0} & \dots & m_{ji} & \dots & m_{jj} & \dots & m_{jn} \\ \vdots & & \vdots & & \vdots & & \vdots \\ m_{n0} & \dots & m_{ni} & \dots & m_{nj} & \dots & m_{nn} \end{pmatrix}$$

- Choose (i, j) such that $|m_{ij}|$ is the maximum non-diagonal element
- For the following matrix, force m_{ij} and m_{ji} to vanish

$$\begin{pmatrix} m_{ii} & m_{ij} \\ m_{ji} & m_{jj} \end{pmatrix}$$

- Propagate the computation effects along the rows and columns

Singular Value Decomposition – Jacobi method

- Major drawback: Jacobi method requires at each step the scanning of $n(n-1)/2$ numbers for one of maximum modulus
  - This can be time consuming for large matrices
- **Cyclic Jacobi method**: select the pairs (i, j) in some cyclic order
- Try the following order (cyclic-by-rows):

  $1-2, 1-3, \ldots, 1-n, 2-3, \ldots, 2-n, 3-4, \ldots (n-1)-n$
- More than one sweep may be needed!
- Although some on-diagonal energy may go off-diagonal at some iterations, the process is known to converge in a small number of sweeps
- *It is not needed to vanish a non-diagonal element completely!*
  - Think in terms of off-diagonal energy going on-diagonal

Singular Value Decomposition – the core operation

- The basic operation is the two-sided rotation of each 2×2 matrix.

$$R(\theta_l)^T \begin{pmatrix} a & b \\ c & d \end{pmatrix} R(\theta_r) = \begin{pmatrix} \Psi_1 & 0 \\ 0 & \Psi_2 \end{pmatrix}$$

  where θ_l and θ_r are the left and right rotation angles, respectively.

- The input 2×2 matrix subject to diagonalization is:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

- A rotation matrix has the following form:

$$\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

- Two issues need to be addressed:

- Calculation of the rotation angles
- Performing the rotations

Singular Value Decomposition – operation budget

- **Calculation of the rotation angles requires**:
  - The evaluation of $\arctan$
- $\arctan$ is a transcendental function
  - Is series expansion appropriate to evaluate $\arctan$?
- **Performing the rotations requires**:
  - The evaluation of $\cos$ and $\sin$
  - Matrix multiplication
- $\cos$ and $\sin$ are transcendental functions
  - Is series expansion appropriate to evaluate $\cos$ and $\sin$?
- Matrix multiplication can be carried out within the standard instruction set

Singular Value Decomposition (cont'd)

- The efficient computation of the rotation parameters is essential.
- The direct two-angle method calculates θ_l and θ_r by computing the inverse tangents of the data elements of M:

$$\theta_{\text{SUM}} = \theta_r + \theta_l = \arctan\left(\frac{c+b}{d-a}\right)$$

$$\theta_{\text{DIFF}} = \theta_r - \theta_l = \arctan\left(\frac{c-b}{d+a}\right)$$

- The two angles, θ_l and θ_r, can be separated from the sum and difference results and applied to the two-sided rotation module to diagonalize M.
- In a typical serial computer, the **calculation of the rotation angles** and **performing the rotations** are both expensive tasks.
- Provide architectural support (define a new instruction and deploy the associated computing unit) for $\arctan$, $\cos$, $\sin$

How to calculate $\arctan(x)$?

- The function $\arctan : (-\infty, \infty) \longrightarrow \left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$
  - Integer representation: we clearly don't like a domain like $(-\infty, \infty)$
  - **An idea!**
    - Calculate $\arctan(x)$ when $|x| \leq 1$
    - Calculate $\text{arccot}(x)$ when $|x| > 1$ and adjust the angle accordingly

- In C using floating point:

```
#include <math.h>
int x, y, angle;

if (x > y)
  angle = arctan( y/x);         /* arctan() returns a float */
else
  angle = PI/2 - arctan( x/y);  /* arctan() returns a float */
```

How to calculate $\arctan(x)$?

- Integer arithmetic required!
  - C standard library (**math.h**): $\arctan()$ is a floating-point function
  - "/" is not a good option to divide integers
  - π is a fractional number

- Implement our own $\arctan()$ routine – what algorithm shall we use?
  - Taylor series expansion about a point – approximation good for 1 point

$$\arctan(x) = x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \dots$$

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

– Piecewise linear approximation with three middle points:

$$\arctan(x) = \begin{cases} 0.644\,x + 0.142 & \text{if } 0.5 < x \leq 1.0, \\ 0.928\,x & \text{if } -0.5 \leq x \leq 0.5, \\ 0.644\,x - 0.142 & \text{if } -1.0 \leq x < -0.5. \end{cases}$$

$\arctan(x)$ – piecewise linear approximation

- The formula using fractional numbers

$$\arctan(x) = \begin{cases} 0.644\,x + 0.142 & \text{if } 0.5 < x \leq 1.0, \\ 0.928\,x & \text{if } -0.5 \leq x \leq 0.5, \\ 0.644\,x - 0.142 & \text{if } -1.0 \leq x < -0.5. \end{cases}$$

- From fractional to integer (assume 12-bit signed integer representation):
  - 1.0 is *represented* as 2^{11} (in fact, as $2^{11} - 1$)
  - 0.928 is represented as $1900 = 76\mathrm{C_h}$
  - 0.644 is represented as $1319 = 527_\mathrm{h}$
  - 0.142 is represented as $291 = 123_\mathrm{h}$
  - 0.5 is represented as $1024 = 400_\mathrm{h}$
  - x is represented as $X = 2^{11}\,x$

$\arctan(x)$ – piecewise linear approximation

- Piecewise linear approximation using integer arithmetic

$$\arctan(X) = \begin{cases} 1319\,X + 291 & \text{if } 1024 < X \leq 2048, \\ 1900\,X & \text{if } -1024 \leq X \leq 1024, \\ 1319\,X - 291 & \text{if } -2048 \leq X < -1024. \end{cases}$$

- $\arctan(X)$ is a signed integer ranging $-1,350,947 \cdots + 1,350,947$, which in hex is $-149\mathrm{D}23_\mathrm{h} \cdots + 149\mathrm{D}23_\mathrm{h}$
  - Homework: how many bits are needed to represent $\arctan(X)$?

- Questions that can be posed:
  - Computing time: software implementation versus hardware implementation
  - Precision of the piecewise linear approximation using integer arithmetic

- Same problem for $\sin(x)$ and $\cos(x)$

Jacobi method – side effects

- It works fine with rectangular matrices, too.

- If the matrix is symmetric, the algorithm finds the eigenvalues.

- Matrix triangularization can be achieved with one-side rotations
  - Upper triangularization with left-side rotations
  - Lower triangularization with right-side rotations

Jacobi method – bibliography

- Professor Richard P. Brent:

  `http://web.comlab.ox.ac.uk/oucl/work/richard.brent/`

- Any textbook on linear algebra

Matrix diagonalization – project requirements

- Build the testbench: the input is a square matrix of integers
- Assume the piecewise linear approximation for $\arctan$, $\sin(x)$, and $\cos(x)$, and determine the maximum error for an approximation with three middle points.
- Implement piecewise linear approximation using integer arithmetic for $\sin$, $\cos$, and $\arctan$ in:
  - software (write C routines)
  - horizontal firmware with two issue slots
  - custom hardware (write VHDL/Verilog)
- Define a new instruction that will return the trigonometric function
  - You must comply with the ARM architecture (you can have at most two arguments and one result per instruction call)

Matrix diagonalization – project requirements

- Rewrite the high-level code and instantiate the new instruction
  - Use assembly inlining
- Diagonalize a square matrix using piecewise linear approximation of trigonometric functions 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
- Estimate the penalty in terms of number of gates for the hardware solution

Lesson 5: COordinate Rotation DIgital Computer (CORDIC)

Motivation

- Assume vector $[x, y]^T$ is being rotated with an angle θ
- Assuming the rotated vector is $[x', y']^T$
- The equation set that describes this rotation is:

$$\begin{cases} x' = x\cos\theta + y\sin\theta \\ y' = y\cos\theta - x\sin\theta \end{cases}$$

- Direct evaluation is computationally demanding
  - Evaluation of trigonometric functions translates to a sequence of multiplications, additions, and memory look-up operations if the common Taylor series expansion is employed.

COordinate Rotation DIgital Computer (CORDIC)

- **CORDIC** is an iterative method performing vector rotations by arbitrary angles using only shifts and additions
  - **Cheap**: only shifts and additions are needed
  - **Sequential**: it is an iterative method

- The iterative method: the rotation angle θ is splitted into a sequence of subrotations of elementary angles $\theta[i]$, where the rotation for iteration i is

$$\begin{cases} x[i+1] = x[i]\cos\theta[i] + y[i]\sin\theta[i] \\ y[i+1] = y[i]\cos\theta[i] - x[i]\sin\theta[i] \end{cases}$$

- The elementary angles $\theta[i]$ are predefined

CORDIC

- Only shifts and additions: rotation angles are restricted so that $\tan\theta[i] = \pm 2^{-i}$
  - Multiplication by the tangent factor is reduced to a shift operation:

$$\begin{cases} x[i+1] = x[i]\cos\theta[i] + y[i]\sin\theta[i] = \cos\theta[i](x[i] + y[i]\tan\theta[i]) \\ y[i+1] = y[i]\cos\theta[i] - x[i]\sin\theta[i] = \cos\theta[i](y[i] - x[i]\tan\theta[i]) \end{cases}$$

$$\begin{cases} x[i+1] = \cos\theta[i](x[i] + 2^{-i}y[i]) \\ y[i+1] = \cos\theta[i](y[i] + 2^{-i}x[i]) \end{cases}$$

- Arbitrary rotation angles can be obtained by performing a series of successively smaller elementary rotations
- The decision at each iteration is which direction to rotate
  - The factor $\cos\theta[i]$ is a constant for the current iteration
  - The product of all these cosine values is a constant = system processing gain

CORDIC

- The angle of a composite rotation is uniquely defined by the sequence of the directions of the elementary rotations
- That sequence can be represented by a decision vector
- The set of all possible decision vectors is an angular measurement system based on binary arctangents
- Conversions between this angular system and any other can be accomplished using an additional adder-subtractor that accumulates the elementary rotation angles at each iteration

$$z[i+1] = z[i] - \sigma[i] \arctan\left(2^{-i}\right)$$

- The elementary angles are supplied by a small look-up table (one entry per iteration), or are hardwired, depending on the implementation

The arctangent table in degrees

$\theta(i) = \arctan(2^{-i})$

Iteration i	Elementary angle degrees
0	45.00
1	26.56
2	14.04
3	7.13
4	3.58
5	1.79
6	0.89
7	0.45

Example: $30.00 \approx 45.00 - 26.56 + 14.04 - 7.13 + 3.58 + 1.79 - 0.89 + 0.45$

CORDIC rotation mode

- The angle accumulator is initialized with the desired rotation angle
- The rotation decision at each iteration is made to diminish the magnitude of the residual angle in the angle accumulator

$$\begin{cases} x[i+1] = x[i] - \sigma[i]2^{-i}y[i] \\ y[i+1] = y[i] + \sigma[i]2^{-i}x[i] \\ z[i+1] = z[i] - \sigma[i]\arctan\left(2^{-i}\right) \end{cases} \quad \text{where} \quad \sigma[i] = \begin{cases} -1 & \text{if } z[i] < 0, \\ +1 & \text{otherwise.} \end{cases}$$

- After n iterations, the result is:

$$\begin{cases} x[n] = A[n](x[0]\cos z[0] - y[0]\sin z[0]) \\ y[n] = A[n](y[0]\cos z[0] - x[0]\sin z[0]) \\ z[n] = 0 \end{cases} \quad \text{where} \quad A[n] = \prod_{i=0}^{n} \sqrt{1 + 2^{-2i}}$$

CORDIC vectoring mode

- The input vector is rotated through whatever angle is necessary to align the result vector with the x axis
- The result of the vectoring operation is a rotation angle and the scaled magnitude of the original vector (the x component of the result)

$$\begin{cases} x[i+1] = x[i] - \sigma[i] 2^{-i} y[i] \\ y[i+1] = y[i] + \sigma[i] 2^{-i} x[i] \\ z[i+1] = z[i] - \sigma[i] \arctan\left(2^{-i}\right) \end{cases} \quad \text{where} \quad \sigma[i] = \begin{cases} -1 & \text{if } y[i] \geq 0, \\ +1 & \text{otherwise.} \end{cases}$$

- After n iterations, the result is:

$$\begin{cases} x[n] = A[n] \sqrt{x_0^2 + y_0^2} \\ y[n] = 0 \\ z[n] = z[0] + \arctan(y[0]/x[0]) \end{cases} \quad \text{where} \quad A[n] = \prod_{i=0}^{n} \sqrt{1 + 2^{-2i}}$$

CORDIC numerical properties

- The CORDIC algorithm produces one bit of accuracy for each iteration
  - Accuracy can be adjusted dynamically by adding or removing iterations

- To preserve N bits of significance in a fixed-point implementation, $\log_2 N$ additional low-order bits are necessary for intermediate values.
  - $N + \log_2 N$-bit word length is needed for N-bit CORDIC precision
  - Example: 12-bit CORDIC precision is guaranteed with a 16-bit wordlength

- Domain of convergence is $-\pi/2 \cdots + \pi/2$
  - Rotation mode: $-\pi/2 \leq \theta \leq +\pi/2$
  - Vectoring mode: $x \geq 0$

- Vectoring mode with a zero input vector: the result is undefined.

How to calculate transcendental functions using CORDIC

- $\boldsymbol{\arctan(y/x)}$
  - Build a vector $[x, y]^T$ with the x and y
  - Initialize $z = 0$ and run CORDIC in vectoring mode
  - After n iterations, $z = \arctan(y/x)$ with n-bit precision

- $\boldsymbol{\arctan(x)}$
  - Build a vector $[1, x]^T$ with the x and y
  - Initialize $z = 0$ and run CORDIC in vectoring mode
  - After n iterations, $z = \arctan(x)$ with n-bit precision

- $\boldsymbol{\cos\theta}$ and $\boldsymbol{\sin\theta}$
  - Build a vector $[1, 0]^T$
  - Initialize $z = \theta$ and run CORDIC in rotation mode
  - After n iterations, $x = \cos\theta$, and $y = \sin\theta$ with n-bit precision

CORDIC – project requirements

- Build the testbench:
  - Values for x and y to calculate $\arctan$
  - values for θ to calculate $\cos$ and $\sin$

- Implement the CORDIC algorithm using integer arithmetic
  - software (write C routines)
  - horizontal firmware with two issue slots
  - custom hardware (write VHDL/Verilog)

- Define a new instruction that will return the trigonometric function
  - You must comply with the ARM architecture (you can have at most two arguments and one result per instruction call)

CORDIC – project requirements

- Rewrite the high-level code and instantiate the new instruction
  - Use assembly inlining
- 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
- Estimate the penalty in terms of number of gates for the hardware solution

Lesson 6: Color Space Conversion

- Trichromatic theory and color spaces
- Why we need to go from one color space to another
- Color Space Conversion – linear mapping / matrix transform
- Upsampling and downsampling color space conversion
- Project requirements

Trichromatic Theory

- According to the Trichromatic Theory, it is possible to match all of the colors in the visible spectrum by appropriate mixing of three primary colors
- Which primary colors are used is not important as long as mixing two of them does not produce the third
- For display systems that emit light, the Red-Green-Blue (RGB) system is used
- The nonlinearity of the CRT monitor is compensated by a nonlinear function to RGB intensities to form *Gamma–Corrected Red*, *Green*, and *Blue* (R'G'B')
- A color space is a mathematical representation of a set of colors
- Several standard color spaces: R'G'B', Y'CC, Y'UV
- Y'CC and Y'UV used by video standards
- Nice book: *A Technical Introduction to Digital Video* by Charles Poynton

Why we need color space conversion

- The space RGB: each value represents a color
- The human eye is less sensitive to color than luminance
- To reduce the storage requirements and/or transmission rate
  - Transmit luminance with full resolution
  - Represent the color information with lower resolution
    - *Reduce* the resolution when converting from RGB representation to Luminance+Color representation
    - *Increase* the resolution when converting from a Luminance+Color representation to RGB representation
- Reduce the resolution: **downsampling**
  - Get rid of samples – which ones?
- Increase the resolution: **upsampling**

– Create new samples – how?

Luminance and Chrominance

- **Luma** signal (represents luminance or brightness):

$$Y' = 0.299R' + 0.587G' + 0.114B'$$

  where R', G', and B' range $[0 \cdots + 1]$

- Luma contains a large fraction of the green information
- Form two *color difference* components with no contribution from luminance:

$$B' - Y' = -0.299R' - 0.587G' + 0.886B'$$

$$R' - Y' = 0.701R = -0.587G' - 0.114B'$$

- Matrix notation (be very curious and calculate the condition number!)

$$\begin{pmatrix} Y' \\ B' - Y' \\ R' - Y' \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.299 & -0.587 & 0.886 \\ 0.701 & -0.587 & -0.114 \end{pmatrix} \cdot \begin{pmatrix} R' \\ G' \\ B' \end{pmatrix}$$

Color Space Conversion – $R'G'B'$-to-$Y'P_BP_R$

- Analog video equipment – $Y'P_BP_R$ are defined as follows:
  - Y' ranges $[0 \cdots +1]$
  - P_B and P_R range $[-0.5 \cdots +0.5]$
- To construct $Y'P_BP_R$ from the basic Y', $(B' - Y')$, and $(R' - Y')$
  - Scale the $(B' - Y')$ row by $\dfrac{0.5}{1 - 0.114} = \dfrac{0.5}{0.886}$
  - Scale the $(R' - Y')$ row by $\dfrac{0.5}{1 - 0.299} = \dfrac{0.5}{0.701}$

$$\begin{pmatrix} Y' \\ P_B \\ P_R \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{pmatrix} \cdot \begin{pmatrix} R' \\ G' \\ B' \end{pmatrix}$$

- Inverse transform: **http://www.poynton.com/ColorFAQ.html**

University of Victoria
British Columbia • Canada

Color Space Conversion – $R'G'B'$-to-$Y'P_BP_R$

- The $R'G'B'$-to-$Y'P_BP_R$ transformation assumes:
  - R', G', and B' range $[0 \cdots +1]$
  - Y' ranges $[0 \cdots +1]$
  - P_B and P_R range $[-0.5 \cdots +0.5]$
  - The matrix contains fractional numbers
- We need to use only integer arithmetic!
  - R', G', and B' can be, for example, 8-bit unsigned integers
  - Y' can also be an 8-bit unsigned integer
  - How many bits do we need to represent the matrix elements?
- Saturating arithmetic is needed!
  - R', G', B', and Y' range $[0 \ldots 255]$, P_B and P_R range $[-128 \ldots 127]$
  - Hardware-based solution: make sure the hardware will saturate the result

– Software-based solution on a 32-bit processor: can we use saturating operations to implement 8-bit saturating arithmetic?

Color Space Conversion – $R'G'B'$-to-$Y'C_BC_R$

- Many standards for digital versions of this matrix

- Recommendation ITU-R BT.601-4 $=$ the international standard for studio-quality component digital video

- Luminance Y':
  - Coded in 8 bits
  - Excursion of 219 and an offset of 16 (range of $[+16 \ldots 235]$)
  - The extremes of the coding range provide headroom and footroom for accomodation of ringing from filters

- Chrominance C_B and C_R
  - Coded in 8 bits
  - Excursion of ± 112 and offset of $+128$ (range of $[+16 \ldots 240]$)

Color Space Conversion – $R'G'B'$-to-$Y'C_BC_R$

- To form $Y'C_BC_R$ from Y', $B' - Y'$, $R' - Y'$ in the range $[0 \cdots + 1]$

$$Y' = 16 + 219Y'$$

$$C_B = 128 + 112\left[\frac{1}{1 - 0.114}(B' - Y')\right]$$

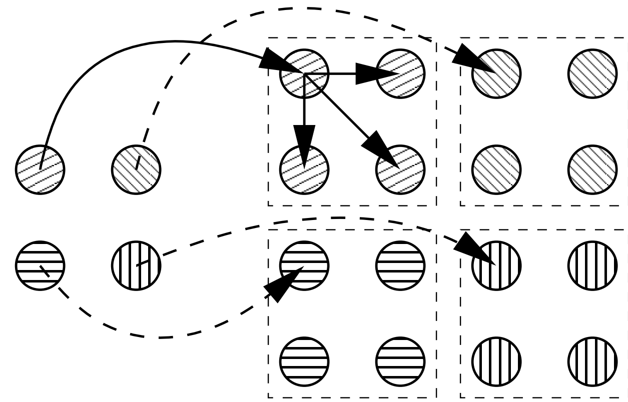
$$C_R = 128 + 112\left[\frac{1}{1 - 0.299}(R' - Y')\right]$$

- Matrix form: scale the rows by the factors 219, 224, and 224

$$\begin{pmatrix} Y' \\ C_B \\ C_R \end{pmatrix} = \begin{pmatrix} 16 \\ 128 \\ 128 \end{pmatrix} + \begin{pmatrix} 65.481 & 128.553 & 24.966 \\ -37.797 & -74.203 & 112.0 \\ 112.0 & -93.786 & -18.214 \end{pmatrix} \cdot \begin{pmatrix} R' \\ G' \\ B' \end{pmatrix}$$

Color Space Conversion – Y'CC-to-R'G'B'

$$\begin{cases} R' = 1.164(Y' - 16) + 1.596(Cr - 128) \\ G' = 1.164(Y' - 16) - 0.813(Cr - 128) \\ \qquad\qquad\qquad\qquad - 0.391(Cb - 128) \\ B' = 1.164(Y' - 16) + 2.018(Cb - 128) \end{cases}$$



Two-dimensional 2–fold upsampling by replication

What we have here from the point of view of computing pattern:

- Y', Cb, Cr, R', G', and B' are 8-bit integers
- For each Cr and Cb *chroma* values there are four *luma* values (because the human eye is less sensitive to color than luminance) – we have to do **upsampling**, e.g., by replication

Color Space Conversion – Y'CC-to-R'G'B' (cont'd)

What we have here from the point of view of computing pattern (cont'd):

- How do we represent fractional numbers in fixed-point? In our case, for example, the largest multiplier is 2.018. Assuming we want a 16-bit representation, 4 (the immediate power of 2 larger than 2.018) is $2^{16} = 65{,}536 = 10000$ h. Then, 2.018 will be $33{,}062 = 8126$ h, and 1.164 will be $19{,}070 = 4A7E$ h.

- Subtraction by a constant that is a power of 2 – it has implications in latency, since the carry should not be propagated all over the word width

59	-	**0011 1011**	-
16		**0001 0000**	
43		**0010 1011**	

Color Space Conversion – Y'CC-to-R'G'B' (cont'd)

What we have here from the point of view of computing pattern (cont'd):

- Multiplication by constant, e.g., 1.164 (= 4A7E h) – we do not have to build the complete partial product matrix, that is, the rows corresponding to zero in the multiplier are discarded
- We can still do tricks: 0111111 = 1000000 - 1

```
                                0 1 0 0 1 1 0 1
4A7E -->        0 1 0 0 1 0 1 0 0 1 1 1 1 1 1 0 x
-----------------------------------------------
                              0 1 0 0 1 1 0 1   +
                            0 1 0 0 1 1 0 1     +
                          0 1 0 0 1 1 0 1       +
                        0 1 0 0 1 1 0 1         +
                      0 1 0 0 1 1 0 1           +
                    0 1 0 0 1 1 0 1             +
              0 1 0 0 1 1 0 1                   +
          0 1 0 0 1 1 0 1                       +
    0 1 0 0 1 1 0 1                             +
-----------------------------------------------
              (perform the addition)
```

```
                                0 1 0 0 1 1 0 1
4A7E -->        0 1 0 0 1 0 1 0 0 1 1 1 1 1 1 0
-----------------------------------------------
                              0 1 0 0 1 1 0 1   −
                  0 1 0 0 1 1 0 1               +
              0 1 0 0 1 1 0 1                   +
          0 1 0 0 1 1 0 1                       +
    0 1 0 0 1 1 0 1
-----------------------------------------------
              (perform the addition)
```

Color Space Conversion – R'G'B'-to-Y'CC

$$\begin{cases} Y' - 16 = +0.257R' + 0.504G' + 0.098B' \\ Cb - 128 = -0.148R' - 0.291G' + 0.439B' \\ Cr - 128 = +0.439R' - 0.368G' - 0.071B' \end{cases}$$

What we have here from the point of view of computing pattern:

- Y', Cb, Cr, R', G', and B' are 8-bit integers
- For each Cr and Cb *chroma* values there are four *luma* values – we have to do **downsampling**
- Downsampling: four Cr (or Cb) values are replaced by a single Cr (or Cb) value
  - Approach 1: discard three values and keep one
  - Approach 2: **calculate the average of the four values**
  - Approach 3: more complex filtering

Color Space Conversion – Upsampling

- Increase the spatial resolution of chrominance signals to double
- Upsampling is carried out on both horizontal and vertical dimensions
  - One Cr (or Cb) value is replaced by four Cr (or Cb) values
  - The resulting image will be four times larger than the initial one
- How to create new pixels?
  - *Programmer with no DSP skills*: replicate the pixel – poor quality of the resulting image
  - *Lazy programmer*: generate the new pixels by linear interpolation – slightly better image quality
  - *DSP-skilled programmer* uses the Filter Theory, since increasing the sampling rate is a filtering problem
- Be very curious and try the last approach!

Color Space Conversion – Downsampling

- Reduce the spatial resolution of chrominance signals to half
- Downsampling is carried out on both horizontal and vertical dimensions
  - Four Cr (or Cb) values are replaced by a single Cr (or Cb) value
  - The resulting image will be four times smaller than the initial one
- How to filter out every other pixel?
  - *Programmer with no DSP skills*: discard every other pixel – poor quality of the resulting image
  - *Lazy programmer*: the resulting pixel is the average of four pixels – slightly better image quality
  - *DSP-skilled programmer* uses the Filter Theory, since reducing the sampling rate is a filtering problem
- Be very curious and try the last approach!

Color Space Conversion – pure-software solution

- Start with *float* type to get a running code quickly
- The best test: apply also the inverse transform to get the initial values

```
float r, g, b, y, cr, cb;

int main( void) {
  for( ... all rows in an image ...)
    for( ... all columns in an image ...) {
      ... read r, g, b ...
      y = 16.0 + 0.257 r + 0.504 g + 0.098 b
      cb = 128.0 - 0.148 r - 0.291 g + 0.439 b
      cr = 128.0 + 0.439 r - 0.368 g - 0.071 b
      ... do this conversion 4 times ...
      ... average four cb/cr values to do downsampling ...
    }
```

```
  exit( 1);
}
```

Color Space Conversion – pure-software solution

- Compile this code on ARM: each float operation is compiled into a large number of instructions (entire routines!)
  - Reason: there is no floating-point unit
- On an embedded platform this is clearly way too slow!
  - Strategy: trade off precision for computing time
- Convert the *float* arithmetic to *integer* arithmetic
  - 0.257 will become an integer
  - 16.0 will also become an integer – think twice if you want to represent the real value 16.0 as integer 16
- **Difficulty**: the large dynamic range to be converted
  - The smallest real value: 0.071
  - The largest value: 128.0

Color Space Conversion – hardware-based solution

- Assume that the pure-software solution is too slow
- Investigate hardware support for computationally-demanding operations
- **What to support in hardware?**
  - The entire matrix transform (all three lines)?
  - One line at a time? In this case we need three separate new instructions
- Limitations due to the architecture of the host processor (ARM in our case)
  - Only two input arguments and one result per instruction are allowed
  - More than two arguments needed? Additional dummy instructions needed to upload extra arguments to the functional unit
  - More than one result needed? Additional dummy instructions needed to donwload the extra result(s) from the functional unit
  - **Packing** the input arguments and/or results when the word-width of the arguments and results is not large (as it is the case in color space conversion)

Color Space Conversion – hardware-based solution

- Assume the packing strategy
- Downsampling can be carried out in hardware, too! Think about that.

```
int r, g, b, y, cr, cb;

int main( void) {
  for( ... all rows in an image ...)
    for( ... all columns in an image ...) {
      ... read r, g, b ...
      ... pack r, g, b ...         << This is overhead! >>
      CALL_HARDWARE ( r, g, b, y, cb, cr)
      ... unpack y, cb, cr ...     << This is overhead! >>
      ... do this conversion 4 times ...
      ... average four cb/cr values to do downsampling ...
    }
```

University of Victoria
British Columbia • Canada

```
  exit( 1);
}
```

Color Space Conversion – project requirements

- Build a testbench containing an image (the image should be large enough in order not to fit into the cache)
- Design a color space conversion algorithm using only integer arithmetic
  - Do not forget that you need saturating arithmetic
- Provide a pure-software solution and estimate its performance
  - Keep an eye on cache misses
- Provide hardware and firmware support for multiplication-by-constant operations
  - Define new instructions driving the new hardware/firmware units
  - Rewrite the code in order to instantiate the new instructions
- Compare the hardware- and firmware-assisted solution with the pure software solution

Lesson 7: Motion estimation

- Motion estimation theory
- Motion estimation hardware support
- Project requirements
- Bibliography:

S. Vassiliadis et al., *The Sum-Absolute-Difference Motion Estimation Accelerator*, in Proceedings of the 24th Euromicro Conference, pp. 559-566, Vasteras, Sweden, August 1998, pp. 559-566.

http://ce.et.tudelft.nl/publicationfiles/474_2_00708071.pdf

Motion estimation

- Similarities between video frames are exploited to achieve high compression rate
- Instead to code (and thus transmit) a new frame, code only the relative movement of the current frame with respect to the previous one
- **Motion estimation** algorithm captures such movement by finding the **best match** of an n-by-n block in a reference frame
- Commonly used metric – **Sum-of-Absolute-Differences** (SAD)
- Motion estimation is performed typically on a block of pixels
- SAD operation is usually considered for 16×16-pixel blocks
- The search area could involve a large number of blocks
- SAD operation can be time consuming

Motion estimation process

- Motion estimation is performed on a set of pixels
- Each frame is divided into blocks of equal size
- For each block in the current frame a search is performed in the reference frame to find the block resembling the current block the most
- The search is limited to a rather small area
  - A search performed over the whole reference frame for each block in the current frame is computationally intensive
  - Movements in video sequences are usually small
- After finding the best match for the current block in the current frame, two elements are stored:
  - A motion vector (displacement relative to the current block)
  - Difference between the two blocks

Motion estimation (cont'd)

- (x, y) is the position of the current block
- (r, s) is the motion vector (the displacement of the current block A relative to the reference block B)
- The computation per each block pair

$$SAD(x, y, r, s) = \sum_{i=0}^{15} \sum_{j=0}^{15} |A(x + i, y + j) - B((x + r) + i, (y + s) + j)|$$

- How many block pairs per frame are analyzed?
  - Motion estimation is the bottleneck in video coding

Motion estimation – software solution

- (x, y) is the position of the current block
- (r, s) is the motion vector (the displacement of the current block A relative to the reference block B)
- Lines 07-09: 1 comparison, 1 branch, 0.5 subtraction, 1 addition

```
01   int A[16][16], B[16][16], diff, sad = 0;
02   int i, j;
03
04   for( i=0; i<16; i++)
05   for( j=0; j<16; j++) {
06     diff = A[x+i][y+j] - B[(x+r)+i][(y+s)+j];
07     if( diff < 0)       /* takes the absolute value */
08       diff -= diff;
09     sad += diff;
10   }
```

Motion estimation – software solution

- The explicit implementation of the absolute operation requires 3.5 operations
- The following code requires only 3 operations
- Penalty: the code size is 1 instruction larger (typically, this is a good trade-off)

```
int A[16][16], B[16][16], diff, sad = 0;
int i, j;

for( i=0; i<16; i++)
for( j=0; j<16; j++) {
  diff = A[x+i][y+j] - B[(x+r)+i][(y+s)+j];
  if( diff < 0)
    sad -= diff;
  else
    sad += diff;
}
```

Motion estimation – software solution

- Operation budget
  - 256 subtractions
  - 256 if-then-else to calculate the absolute value
  - 256 additions/subtractions
  - 256 comparisons
  - 256 incrementations
  - 256 branch operations
- Large number of operations per block pair
- Large number of true dependencies – the code is sequential
  - Would software pipelining, loop unrolling, etc. help?
- **Hardware** support is needed
- The code is sequential $\longrightarrow$ firmware assist is not likely to provide improvement

Improving the software solution – loop unrolling

```
int A[16][16], B[16][16], diff1, diff2, sad = 0;
int i, j;

for( i=0; i<16; i++)
for( j=0; j<16; j+=2) {
  diff1 = A[x+i][y+j] - B[(x+r)+i][(y+s)+j];
  diff2 = A[x+i][y+j+1] - B[(x+r)+i][(y+s)+j+1];
  if( diff1 < 0)
    sad -= diff1;
  else
    sad += diff1;
  if( diff2 < 0)
    sad -= diff2;
  else
    sad += diff2;
}
```

University of Victoria

British Columbia • Canada

Improving the software solution – loop unrolling

- Generally it is not possible to execute two branch operations in parallel
- Would guarded operations help executing two branch operations in parallel?
- Parallelism: Lines 07 and 08
- Load and store operations can be overlapped to some extend
- The 'for' loops generate only $16 \times 8 = 128$ branch operations
  - This is half of the initial branch count
  - Penalty: double code size that might induce instruction cache misses
- Conclusion: loop unrolling does not provide significant improvement (if any)
- Homework: analyze software pipelining

Motion estimation – hardware solution

- **What to implement in hardware?**
  - A single absolute-difference operation and do the accumulation in software?
  - A sum of 256 absolute-differences (that is, everything in hardware)?
  - A partial sum of absolute-differences and perform the rest of the accumulation in software?
- Host processor architectural constraints – ARM example:
  - Two 32-bit arguments per instruction call
  - One 32-bit result per instruction call
  - Are non-reentrant instructions allowed?
- Assume for the sake of presentation that
  - Each pixel is represented on an 8-bit signed integer
  - Four pixels fit into a 32-bit register (argument in our case)
- The hardware will calculate the sum-of-absolute-differences for four pixel pairs

Motion estimation – reentrant or non-reentrant unit?

- The new unit is called Sum-of-Absolute-Differences (SAD)
- **Reentrant SAD**
  - The unit does not have state
  - The output depends only on the inputs
  - The function to be implemented is of zeroth order
    * SAD can be a combinational circuit
    * A higher order circuit can be used, but the function to be implemented is still of zeroth order
  - **The accumulation is done in software**
- **Non-reentrant SAD**
  - The unit has state
  - The output depends on the inputs and on the previous state
  - The function to be implemented is of order greater than zero

- **The accumulation is done in hardware**

- How is this problem solved in a Multiply-and-ACumulate unit?

Motion estimation – hardware solution

- Sum-of-Absolute-Differences (SAD) instruction

  SAD Rs1, Rs2, Rt

  where

  - Rs1 (source register 1) contains four pixels from the current frame
  - Rs2 (source register 2) contains four pixels from the reference frame
  - Rt (target register) contains the sum-of-absolute-differences for these four pixel pairs

- The software routine will be rewritten using the new instruction

- We will assume that four pixels are packed into one 32-bit integer (the column index ranges $[0 \ldots 3]$)

```
int A[16][4], B[16][4];
```

- If the pixels are not packed, then we must pack them beforehand if we want to use the SAD instruction.

Motion estimation – reentrant SAD

- The routine using the new SAD instruction is presented below
- The SAD instruction is called 64 times per 16×16 block
- To find the performance of the code below we need to know the SAD latency

```
int A[16][4], B[16][4], sad = 0;
register int Rs1, Rs2, Rt, i, j;

for( i=0; i<16; i++)
for( j=0; j<4; j++) {
  Rs1 = A[i][j];
  Rs2 = B[i][j];
  __asm__( "SAD %1, %2, %0" : "=r" (Rt) : "r" (Rs1), "r" (Rs2));
  sad += Rt;
}
```

Motion estimation – non-reentrant SAD

- The accumulation is done inside the SAD unit
- A reset instruction for SAD needed
- The interrupts should be disabled during the 'for' loops execution

```
int A[16][4], B[16][4], sad = 0;
register int Rs1, Rs2, Rt, i, j;

__asm__( "RESET_SAD %1, %2, %0" : "=r" (Rt) : "r" (Rs1), "r" (Rs2));
for( i=0; i<16; i++)
for( j=0; j<4; j++) {
  Rs1 = A[i][j];
  Rs2 = B[i][j];
  __asm__( "SAD %1, %2, %0" : "=r" (Rt) : "r" (Rs1), "r" (Rs2));
}
sad = Rt;
```

Motion estimation – project requirements

- Build the testbench: two 2D arrays representing the *current* and *reference* frame
- Determine the performance of the pure software solution
- Build a 4-pixel-pair SAD unit in hardware and determine its latency
  - Most of the effort will be directed here
- Try also a non-reentrant hardware unit and compare it against the reentrant counterpart
- Rewrite the high-level code and instantiate the new instruction
  - Use assembly inlining
- Determine the performance of the hardware-based solution
- Determine the penalty in terms of silicon area of the hardware-based solution

Lesson 8: Digital filtering

- Theory of digital filtering
- FIR and IIR
- Project requirements
- Bibliography:

John G. Proakis and Dimitris G. Manolakis, *Digital Signal Processing. Principles, Algorithms, and Applications*, Third Edition, Prentice Hall, 1996.

FIR and IIR realizations

- FIR is described by the difference equation

$$y(n) = \sum_{k=0}^{N} h(k)x(n-k)$$

- It can be realized with a non-recursive structure by implementing this equation
- It can also be realized by a recursive structure

  H.H. Dam, S. Nordebo, K.L. Teo, and A. Cantoni, *Design of Linear Phase FIR Filters with Recursive Structure and Discrete Coefficients*, in Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, vol. III, pp. 1269-1272, Seattle, USA, May 1998.

- Theory of digital filtering
  - FIR is stable if implemented as a sum-of-products

- Can a FIR become unstable if implemented recursively?
- IIR is not stable if the poles are outside the unit circle

Quantization of filter coefficients

- The accuracy with which filter coefficients can be specified is limited
  - in software: by the word length of the processor
  - in hardware: by the silicon area and latency
- Since the coefficients are not exact, the poles and zeros of the system function will be different from the desired poles and zeros
- The resulting filter has a frequency response that is different from the desired response
- Problems in digital filtering
  - Quantization of filter coefficients
  - Round-off noise in multiplication
  - Overflow in addition
  - Limit cycles

Limit-cycle oscillations in recursive systems

- Finite-precision arithmetic often causes periodic oscillations to occur in the output, even when the input sequence is zero or some nonzero constant value
- Such oscillations in recursive systems are called **limit cycles**
- Limit cycles are directly attributable to:
  - Round-off errors in multiplication
  - Overflow errors in addition
- Classical example: single-pole system

$$y(n) = ay(n-1) + x(n)$$

$$Y(z) = aY(z)z^{-1} + X(z) \Longrightarrow H(z) = \frac{Y(z)}{X(z)} = \frac{z}{z-a}$$

University of Victoria
British Columbia • Canada

- The pole is at $z = a$

Limit-cycle oscillations in recursive systems

- The system implemented with infinite precision:

$$y(n) = ay(n-1) + x(n)$$

- The response to the unit step, $x(n) = \delta(n)$, is $y(n) = a^n$
- The system is stable if the pole is inside the unit circle:

$$\lim_{n\to\infty} a^n = 0 \qquad when a < 1$$

- Assume fixed-point signed-magnitude representation
  - Four bits for magnitude plus a sign bit
  - Rounding that follows multiplication is done upward

- Assume for the sake of presentation that $y(-1) = 0$ and

$$x(n) = 0.1111_2\,\delta(n) = \frac{15}{16}\,\delta(n) = \begin{cases} \frac{15}{16} & \text{if } n = 0, \\ 0 & \text{if } n \neq 0. \end{cases}$$

Limit-cycle oscillations in recursive systems

- Assume $a = 3/4 = 0.1100_2$

$$y(0) = y(-1) \times a + x(0) = 0.0000_2 \times 0.1100_2 + 0.1111_2 = 0.1111_2$$

$$y(1) = y(0) \times a + x(1) = 0.1111_2 \times 0.1100_2 + 0.0000_2 = 0.10110100_2 \approx 0.1011_2$$

$$y(2) = y(1) \times a + x(2) = 0.1011_2 \times 0.1100_2 + 0.0000_2 = 0.10000100_2 \approx 0.1000_2$$

$$y(3) = y(2) \times a + x(3) = 0.1000_2 \times 0.1100_2 + 0.0000_2 = 0.01100000_2 \approx 0.0110_2$$

$$y(4) = y(3) \times a + x(4) = 0.0110_2 \times 0.1100_2 + 0.0000_2 = 0.01001000_2 \approx 0.0101_2$$

$$y(5) = y(4) \times a + x(5) = 0.0101_2 \times 0.1100_2 + 0.0000_2 = 0.00111100_2 \approx 0.0100_2$$

$$y(6) = y(5) \times a + x(6) = 0.0100_2 \times 0.1100_2 + 0.0000_2 = 0.00110000_2 \approx 0.0011_2$$

$$y(7) = y(6) \times a + x(7) = 0.0011_2 \times 0.1100_2 + 0.0000_2 = 0.00100100_2 \approx 0.0010_2$$

$$y(8) = y(7) \times a + x(8) = 0.0010_2 \times 0.1100_2 + 0.0000_2 = 0.00011000_2 \approx 0.0010_2$$

$$y(9) = y(8) \times a + x(9) = 0.0010_2 \times 0.1100_2 + 0.0000_2 = 0.00011000_2 \approx 0.0010_2$$

Limit-cycle oscillations in recursive systems

- Assume $a = -3/4 = 1.1100_2$

$$y(0) = y(-1) \times a + x(0) = 0.0000_2 \times 1.1100_2 + 0.1111_2 = 0.1111_2$$

$$y(1) = y(0) \times a + x(1) = 0.1111_2 \times 1.1100_2 + 0.0000_2 = 0.10110100_2 \approx 0.1011_2$$

$$y(2) = y(1) \times a + x(2) = 0.1011_2 \times 1.1100_2 + 0.0000_2 = 1.10000100_2 \approx 1.1000_2$$

$$y(3) = y(2) \times a + x(3) = 1.1000_2 \times 1.1100_2 + 0.0000_2 = 0.01100000_2 \approx 0.0110_2$$

$$y(4) = y(3) \times a + x(4) = 0.0110_2 \times 1.1100_2 + 0.0000_2 = 1.01001000_2 \approx 1.0101_2$$

$$y(5) = y(4) \times a + x(5) = 1.0101_2 \times 1.1100_2 + 0.0000_2 = 0.00111100_2 \approx 0.0100_2$$

$$y(6) = y(5) \times a + x(6) = 0.0100_2 \times 1.1100_2 + 0.0000_2 = 1.00110000_2 \approx 1.0011_2$$

$$y(7) = y(6) \times a + x(7) = 1.0011_2 \times 1.1100_2 + 0.0000_2 = 0.00100100_2 \approx 0.0010_2$$

$$y(8) = y(7) \times a + x(8) = 0.0010_2 \times 1.1100_2 + 0.0000_2 = 1.00011000_2 \approx 1.0010_2$$

$$y(9) = y(8) \times a + x(9) = 1.0010_2 \times 1.1100_2 + 0.0000_2 = 0.00011000_2 \approx 0.0010_2$$

University of Victoria
British Columbia • Canada

Limit-cycle oscillations in recursive systems

- When the pole is positive, the output sequence reaches a constant value ($1/8$ for $a = 3/4$)
- When the pole is negative, *the output sequence oscillates* between positive and negative values ($\pm 1/8$ for $a = -3/4$)
- These limit cycles occur as a result of the round-off errors in multiplications
- When the input sequence $x(n)$ to the filter becomes zero, the output of the filter then, after a number of iterations, enters into the limit cycle
- The output remains into the limit cycle until another input of sufficient size is applied that drives the system out of the limit cycle
- The amplitudes of the output during a limit cycle are confined to a range of values that is called the **dead band** of the filter

Limit-cycle oscillations in recursive systems

- The dead band for a single-pole filter is defined by:

$$\frac{2^{-b-1}}{1-|a|}$$

  where b is the number of bits (exclusive of sign) to represent fixed-point numbers

- $a = 0$ then $y(n) = x(n)$ (no selectivity at all) – smallest dead band, which is actually *zero* since $2^{-b-1} \approx 0$ for b bits of representation

- To increase selectivity, we move the pole close to the unit circle – the dead band increases
  - For $b = 4$ and $|a| = 1/2$, the dead band ranges $[-1/16 \cdots + 1/16]$
  - For $b = 4$ and $|a| = 3/4$, the dead band ranges $[-1/8 \cdots + 1/8]$
  - For $b = 4$ and $|a| = 7/8$, the dead band ranges $[-1/4 \cdots + 1/4]$
  - For $b = 6$ and $|a| = 7/8$, the dead band ranges $[-1/16 \cdots + 1/16]$

Limit-cycle oscillations in recursive systems

- The limit-cycle behavior in a two-pole filter is much more complex and larger variety of oscillations can occur
- A two-pole filter is described by the following difference equation

$$y(n) = a_1 y(n-1) + a_2 y(n-2) + x(n)$$

- The dead band for a two-pole filter with complex-conjugate poles:

$$\frac{2^{-b-1}}{1 - |a_2|}$$

- The dead band depends only on $|a_2|$
- a_1 determines the frequency of oscillation

Limit-cycle oscillations in recursive systems

- Multiplication: truncation instead of rounding can eliminate many, although not all, of the limit cycles:

  T. Claasen, W. Mecklenbrauker, and J. Peek, *Second-Order Digital Filter with Only One Magnitude-Truncation Quantizer and Having Practically No Limit Cycles*, in Electronics Letters, Vol. 9, 1973.

- Major drawback when using truncation instead of rounding: truncation results in a biased error when using 2's complement representation

- When using signed-magnitude representation, truncation error is symmetric about zero

- The biased error is undesirable in digital filter implementation

- ARM (and most processors) uses 2's complement representation

Limit cycles due to overflows in addition

- An overflow in addition occurs when the sum exceeds the processor's word size
- When $x(n) = \delta(n)$, there are no limit cycles due to overflows in addition for a single-pole filter (one of the operands is $x(n)$ which is zero for $n > 1$
- Two-pole filter:

$$y(n) = a_1 y(n-1) + a_2 y(n-2) + x(n)$$

- A necessary and sufficient condition for ensuring that no zero-input overflow limit cycles occur (2's complement fractional numbers are assumed):

$$|a_1| + |a_2| < 1$$

This condition is very restrictive!

Limit cycles due to overflows in addition

- Effective remedy for curing the problem of overflow oscillations is to use saturating addition (assuming that saturation occurs infrequently)

- Drawback: saturation arithmetic causes signal distorison

- Good strategy: scale the input signal and the unit sample response such that overflow becomes a rare event.

- Assuming that $x(n)$ is upper bounded by A_x, a necessary and sufficient condition to prevent overflow is:

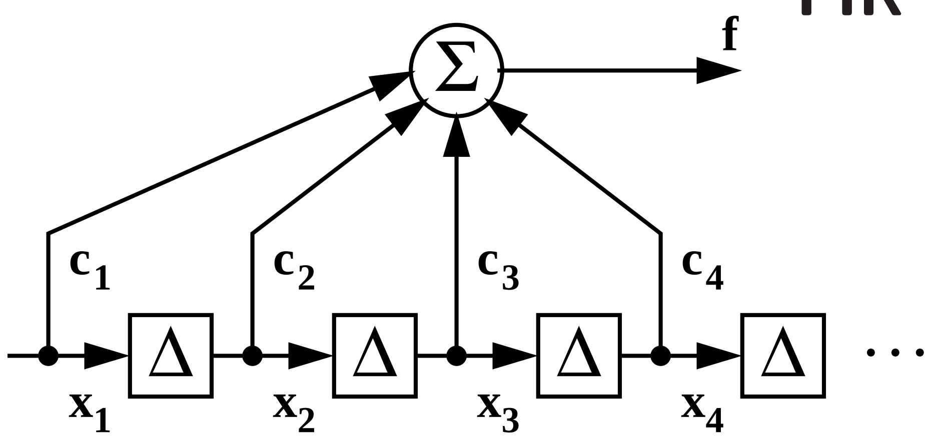
$$A_x < \frac{1}{\sum_{m=-\infty}^{\infty} |h(m)|}$$

University of Victoria
British Columbia • Canada

- For an FIR filter:

$$A_x < \frac{1}{\sum_{m=0}^{M} |h(m)|}$$

FIR filters



$$f = \sum_{i=1}^{4} c_i \cdot x_i$$

- A sum-of-products is to be computed
- Let us assume that
  - Samples are represented on 8-bit signed integers
  - Coefficients are represented on 8-bit unsigned integers
  - Each product (which is represented on a 16-bit signed integer) is quantized to an 8-bit signed integer – this means that right-shifting with rounding/truncation has to be implemented
- Multiply-and-ACumulate (MAC) with shifting and rounding/truncation has to be implemented in hardware/firmware

FIR filters (cont'd)

- Example: $0.7 \times 0.5 = 0.35$
- The maximum signed value $+1.0$ corresponds to 128. Thus, 0.7 is represented as 90
- The maximum unsigned value 1.0 corresponds to 256. Thus 0.5 is represented as 128
- 90 $\times$ 128 = 11520 = 2d00 h = 0010 1101 0000 0000 b
- Right-shifting with 8 positions and rounding:

  0010 1101 b + 0 b = 0010 1101 b = 45
- 128 corresponds to the maximum signed value $+1.0$. Thus, 45 represents

  45 / 128 = 0.35

Digital filtering – project requirements

- Assume a recursive realization for the FIR filter
- Check the limit cycles when using truncation instead of rounding
  - In software use the 2's complement representation
  - In hardware use the sign-magnitude representation
- Is the Multiply-and-ACumulate (MAC) of the ARM processor effective in reducing the limit cycles? What kind of rounding/truncation (if any) does the ARM processor employ?
- Scale the input sequence in order to avoid overflow limit cycles
- Provide a pure-software solution and specify:
  - Its performance (cycle count)
  - The dead band for the particular pole you consider

Digital filtering – project requirements

- Implement in hardware a MAC unit using truncation. If possible, use the sign-magnitude representation instead of 2's complement representation
  - Instantiate the new MAC instruction using assembly inlining
- For the hardware-based solution specify also:
  - The latency of the MAC unit
  - Its performance (cycle count)
  - The dead band for the particular pole you consider
- Determine the penalty in terms of silicon gates for the hardware-based solution

Lesson 9: RSA Cryptography

- RSA: invented by Rivest, Shamir, and Adleman in 1978
- Motivation
  - Secure transmissions over wireless channels
- RSA Cryptography background
- Why is difficult to implement the RSA algorithm
- Project requirements

RSA Cryptography – background

- It uses keys with the length ranging from 512 to 2048 bits
- RSA is based on two distinct odd prime numbers P and Q
- These prime numbers are used to generate two so-called key-pair values:
  - **Public key-pair** (E, PQ) used to encrypt data
  - **Private key-pair** (D, PQ) used to decrypt data
- D, E, and M are very long integers of 512 - 2048 bits

RSA Cryptography – the algorithm

1. Find P and Q, two large (e.g., 1024-bit) prime numbers.

2. Choose E such that: $E > 1$, $E < PQ$, and E and $(P-1)(Q-1)$ are relatively prime (they have no prime factors in common). E does not have to be prime, but it must be odd. $(P-1)(Q-1)$ can't be prime because it's an even number.

3. Compute D such that $(DE-1)$ is evenly divisible by $(P-1)(Q-1)$.

   - Mathematicians write this as $DE = 1(\mod (P-1)(Q-1))$
   - The number D is called the multiplicative inverse of E.

   This is easy to do – simply find an integer X which causes

$$D = (X(P-1)(Q-1)+1)/E$$

   to be an integer, then use that value of D.

University of Victoria
British Columbia • Canada

RSA Cryptography – the algorithm

4. The encryption function is

$$C = T^E \mod PQ$$

where C is the ciphertext (a positive integer), T is the plaintext (a positive integer). The message being encrypted, T, must be less than the modulus, PQ.

5. The decryption function is

$$T = C^D \mod PQ$$

where C is the ciphertext (a positive integer), T is the plaintext (a positive integer).

RSA Cryptography – the algorithm

- Your *public key* is the pair (PQ, E).
- Your *private key* is the number D (reveal it to no one).
- The product PQ is the modulus (often called N in the literature).
- E is the public exponent.
- D is the secret exponent.
- You can publish your public key freely, because there are no known easy methods of calculating D, P, or Q given only (PQ, E) (your public key).
- If P and Q are each 1024 bits long, the sun will burn out before the most powerful computers can factor your modulus into P and Q.

RSA Cryptography – an example

- Choose the **first prime number** (destroy this after computing E and D):

$$P = 61$$

- Choose the **second prime number** (destroy this after computing E and D):

$$Q = 53$$

- Calculate the **modulus** (give this to others)

$$PQ = 3233$$

- Choose the **public exponent** (give this to others)

$$E = 17$$

- Calculate the **private exponent** (keep this secret!)

$$D = 2753$$

RSA Cryptography – an example

- Your public key is (E, PQ). Your private key is D.
- The encryption function is:

$$\text{encrypt}(T) = (T^E) \mod PQ = T^{17} \mod 3233$$

- The decryption function is:

$$\text{decrypt}(C) = (C^D) \mod PQ = C^{2753} \mod 3233$$

- To encrypt the plaintext value 123, do this:

$$\text{encrypt}(123) = 123^{17} \mod 3233 =$$
$$= 337587917446653715596592958817679803 \mod 3233 = 855$$

University of Victoria
British Columbia • Canada

- To decrypt the ciphertext value 855, do this:

$$\text{decrypt}(855) = 855^{2753} \mod 3233 = 123$$

RSA Cryptography – problems

- Arithmetic operations on very long integers: T^E and S^D cannot be computed using common techniques
  - Ideally: suport long-word arithmetic in hardware
  - The bandwidth from register file to functional units is limited and cannot be increased easily
- True data dependencies
  - Rewriting the algorithm to expose the parallelism
  - Collapsing operations and suport the compound in hardware
- Expensive modular and multiplication operations
  - Montgomery Modular Multiplication (MMM)
  - Montgomery Modular Exponentiation (MME)

How to calculate the value of $855^{2753} \mod 3233$

- We know that $2753 = 101011000001_2$, therefore

$$2753 = 1 + 2^6 + 2^7 + 2^9 + 2^{11} = 1 + 64 + 128 + 512 + 2048$$

- Consider this table of powers of 855:

$$\begin{aligned}
855^1 &= 855 \pmod{3233} \\
855^2 &= 367 \pmod{3233} \\
855^4 &= 367^2 \pmod{3233} = 2136 \pmod{3233} \\
855^8 &= 2136^2 \pmod{3233} = 733 \pmod{3233} \\
855^{16} &= 733^2 \pmod{3233} = 611 \pmod{3233} \\
855^{32} &= 611^2 \pmod{3233} = 1526 \pmod{3233} \\
855^{64} &= 1526^2 \pmod{3233} = 916 \pmod{3233}
\end{aligned}$$

How to calculate the value of $855^{2753} \mod 3233$

- The table of powers of 855 (cont'd):

$$
\begin{aligned}
855^{128} &= 916^2 \pmod{3233} = 1709 \pmod{3233} \\
855^{256} &= 1709^2 \pmod{3233} = 1282 \pmod{3233} \\
855^{512} &= 1282^2 \pmod{3233} = 1160 \pmod{3233} \\
855^{1024} &= 1160^2 \pmod{3233} = 672 \pmod{3233} \\
855^{2048} &= 672^2 \pmod{3233} = 2197 \pmod{3233}
\end{aligned}
$$

How to calculate the value of $855^{2753} \mod 3233$

- Given the above, we know this:

$$
\begin{aligned}
&855^{2753} \pmod{3233} = \\
&= 855^{1+64+128+512+2048} \pmod{3233} = \\
&= 855^{1} \times 855^{64} \times 855^{128} \times 855^{512} \times 855^{2048} \pmod{3233} = \\
&= 855 \times 916 \times 1709 \times 1160 \times 2197 \pmod{3233} = \\
&= 794 \times 1709 \times 1160 \times 2197 \pmod{3233} = \\
&= 2319 \times 1160 \times 2197 \pmod{3233} = \\
&= 184 \times 2197 \pmod{3233} = \\
&= 123 \pmod{3233} = \\
&= 123
\end{aligned}
$$

How to calculate the value of $855^{2753} \mod 3233$

- The modulus is not changed frequently, thus the table of powers can be computed off-line
- What is the size of this table of powers?
- The table of powers is too large for an embedded system
- Many techniques to calculate the modular exponentiation have been proposed
  - Modular multiplication is the core of modular exponentiation
- Montgomery arithmetic – recommended literature:

  John Fry and Martin Langhammer, *RSA & Public Key Cryptography in FPGAs*, Altera Corporation.

Modular exponentiation

- A common way: the **multiply and square algorithm**

$$Z = X^E \mod M \qquad \text{where } E = \sum_{i=0}^{n-1} e_i 2^i$$

1. $Z_0 = 1$, and $P_0 = X$
2. FOR $i = 0$ to $n - 1$ LOOP
3. $\quad P_{i+1} = P_i^2 \mod M$
4. $\quad$ IF $e_i = 1$ THEN $Z_{i+1} = Z_i \cdot P_i \mod M$ ELSE $Z_{i+1} = Z_i$
5. END FOR

Modular exponentiation

- The multiply and square algorithm is a running accumulation of squaring and multiplication steps
- At each stage the *modulo* function is performed to keep any intermediate variables within the integer range of M
- A second option is to allow the intermediate variables to grow and perform the *modulo* function as a single final operation
- The first option is typically more desirable as it will keep the multiplication functions down to a practical bit width
- The brute force approach in implementing the *modulo* function involved a divide operation to discover the remainder
- Division is expensive and should be avoided

- The efficiency of the modular multiplier used in the multiply and square algorithm is key to the performance of RSA-based crypto systems.

RSA Cryptography – project requirements

- Determine the word length needed to implement the example presented in class
  - If word-length is less than 32 bits, then you can implement the RSA algorithm within the standard instruction set
  - If word-length is greater than 32 bits, then you need to write routines to implement long-word arithmetic
- Provide a pure-software solution and determine its performance (cycle count)
  - The look-up table (LUT) will be stored into memory
  - How large this LUT should be? Will cache misses be encountered?
  - Try to implement the Montgomery exponentiation in software
- Try a firmware solution
  - The code is sequential – any improvements possible?
  - The firmware engine can be geared to implement long-word arithmetic

RSA Cryptography – project requirements

- Support computation in hardware
  - Try to implement the modular operations on powers of the ciphertext value in parallel
  - Add the results of these modular operations using a multi-operand adder
  - Try to implement the Montgomery exponentiation in hardware
- Specify how many gates are needed to support all or part of the RSA algorithm in hardware