Pulse Code Modulation 11

11.1 OVERVIEW

Pulse code modulation (PCM) is a method of digitizing or quantizing an analog waveform that is used primarily in the transmission of speech signals, for example, in telephone communication. As in any analog-to-digital (A/D) conversion, the quantization process produces an estimate of the signal sample, possibly introducing an error into the digital representation because of the finite number of bits available to represent the value. In theory, this error can be made insignificant by representing the estimate with a large number of bits (high precision). In practice, however, there must be tradeoff between the amount of error and the size of the data representation. The goal is to quantize the data in the smallest number of bits that results in a tolerable error. In the case of speech signals, a linear quantization with 13 or 14 bits is the minimum required to produce a digital representation of the full range of speech signals accurately.

The number of bits required is reduced to eight in the CCITT recommendation G.711 by exploiting a nonlinear characteristic of human hearing. The human ear is more sensitive to quantization noise in small signals than to noise in large signals. G.711 applies a non-uniform (logarithmic) quantization function to adjust the data size in proportion to the input signal. Thus, smaller signals are approximated with greater precision.

Two quantization functions, or encoding laws, are defined by G.711: μ-law and A-law. In most cases, the United States and Japan use μ-law, whereas Europe uses A-law. The ADSP-2100 implementations of PCM encoder and decoder for both laws are provided in this chapter. In each case, algorithmic versions of the encoder and decoder are provided. In theory, the decoder could also be implemented by table lookup, which provides faster conversion at the cost of additional memory. This implementation is straightforward and is not described here.

In practice, an inexpensive codec is often used to perform the A/D conversion. The following routines accept encoded speech samples from a codec and generate a linear sample (decoding), and prepare a linear sample for output to a codec (encoding).

11.2 PULSE CODE MODULATION USING μ-LAW

Rather that taking the logarithm of the linear input directly, which can be difficult, μ-law PCM matches a logarithmic curve with a piecewise linear approximation. Eight straight line segments along the curve produce a close approximation to the logarithmic function. Each of these lines is called a segment. A sample value is represented by its segment and its position within the segment.

The CCITT recommendation provides a μ-law conversion table. This table has several regular characteristics, however, so that the conversion can be implemented without storing the entire table. The PCM value is in signed-magnitude format, so the conversion table for negative numbers is the same as for positive numbers except for the sign. In addition, adding 33 to the segment endpoints produces boundaries at even powers of two.

The format of the μ-law PCM 8-bit word consists of three parts. The most significant bit (MSB) is the sign bit, the next three bits contain the segment number, and the last four bits indicate the position within the segment. All bits of the number are inverted from their actual values to increase the density of 1s, (because speech is typically low-energy) a property that can be used by error-correcting circuitry on transmission lines.

11.2.1 μ-Law PCM Encoder

The ADSP-2100 μ-law encoder subroutine, shown in Listing 11.1, is based on the CCITT recommendation G.711. The only deviation from G.711 is the input format. Because the ADSP-2100 is optimized for full fractional numbers (1.15 format), the subroutine accepts input in 1.15 format, instead of the integer format specified in G.711. To use this routine with integer input values, you would shift the input two bits to the left, maintaining the sign, before calling the routine.

The *u_compress* routine adds 132 (33 shifted left two bits) to the absolute value of the input, then normalizes the result. The most significant non-sign bit is zeroed by exclusive-ORing it with H#4000. The position within the segment is equal to the six MSBs of this number (the two MSBs are zero, giving the four bits). A 10-bit right shift moves these bits to the proper position.

The sign of the output word is generated by testing the sign of the original input, which is stored in the AS flag of the ASTAT register by the absolute value instruction at the start of the routine. The AR register is cleared to all zeros if the sign is positive or set to H#4000 if the sign is negative. This value is shifted to the right seven bits and ORed with the position bits.

The segment number is generated by adding seven to the number of bits the input value was shifted for normalization (stored in the SE register). This number is shifted to the left four bits to move it into the proper position to be ORed with the sign and position bits.

The last step in the encoding process is to invert all bits. This is accomplished with the NOT instruction. The eight LSBs of the AR register hold the encoded PCM value.

```
.MODULE/ROM  u_compression;
{
             Linear to u-law Compression Subroutine

             Calling parameters:
                AR = Input linear value

             Returns with:
                AR = 8-bit u-law value (with all bits inverted)

             Altered Registers:
                AX0,AY0,AR,SE,SR

             Computation Time:
                18 cycles
}

.ENTRY    u_compress;

u_compress:  AR=ABS AR;                        { Absolute value }
             AY0=132;
             AR=AR+AY0;
             SR1=32636;                        {This instruction and the next can}
             IF AV AR=PASS SR1;                {be removed if input ≤ 32636}
             SE=EXP AR (HI);
             SR=NORM AR (LO);                  { Normalize input }
             AY0=H#4000;
             AR=SR0 XOR AY0;                   { clear NMSB }
             SR=LSHIFT AR BY -10 (LO);         { position bits }
             AX0=SE, AR=PASS AY0;
```

(listing continues on next page)

```
            IF POS AR=PASS 0;
            SR=SR OR LSHIFT AR BY -7 (LO);    { sign bit }
            AY0=7;
            AR=AX0+AY0;
            SR=SR OR LSHIFT AR BY 4 (LO);     { segment bits }
            AR=NOT SR0;                       { Invert all bits }
            RTS;
.ENDMOD;
```

Listing 11.1 μ-Law Encoder

11.2.2 μ-Law PCM Decoder

The μ-law PCM decoder expands data received from a transmission line or a codec to the linear domain. The *u_expand* routine decodes a PCM value, requiring 18 cycles to produce a result.

As in the encoder, the only deviation from G.711 is the format of the linear output data. Because the ADSP-2100 is optimized for full fractional data, the output is in 1.15 format, rather than the integer format described in G.711. If integer values are required, you should arithmetically shift the decoder output two bits to the right or remove the shift-by-2 instruction at the end of each routine.

The *u_expand* routine, shown in Listing 11.2, masks out the upper eight bits of the PCM input value and inverts all of the remaining lower eight bits. The segment number is moved, in integer format, from the input word to the SE register. To determine the sign of the input number, H#FF80 is added to the input value. If the sign bit is set, a number greater than zero results; otherwise, the result is a number less than zero.

Control passes to the *negval* block if the input number was negative or *posval* block if it was positive. In the *negval* block the input number (position bits only, no sign or segment bits) is added to itself, effectively shifting the value one bit to the left, then added to 33. The value is shifted by the segment number and stored in the SE register. The shifted value is subtracted from 33 (the same as subtracting 33 from the shifted value and negating the result). Then the number is shifted two bits to the left to place it in 1.15 format.

In the *posval* block, the input number (position bits only) is added to itself, effecting a one-bit left shift. After 33 is added to this value, the result is shifted by the segment number, in the SE register. Then 33 is subtracted from the value, and the result is shifted into 1.15 format, producing a linear value.

```
.MODULE/ROM  u_law_expansion;
{
    This routine determines the 14-bit linear PCM value (right-
    justified) from the 8-bit (right-justified) log (u-law)
    value.

             Calling parameters:
                AR = 8-bit u-law value

             Return values:
                AR = 16-bit linear value (right-justified)

             Altered Registers:
                AR,AF,AX0,AY0,SR,SE

             Computation Time:
                17 Cycles

}

.ENTRY    u_expand;

u_expand: AY0=H#FF;               { mask unwanted bits }
          AF=AR AND AY0, AX0=AY0;
          AF=AX0 XOR AF;          { invert bits }
          AX0=H#70;
          AR=AX0 AND AF;          { isolate segment bits }
          SR=LSHIFT AR BY -4 (LO); { shift to LSBs }
          SE=SR0, AR=AR XOR AF;   { remove segment bits }
          AY0=H#FF80;
          AF=AR+AY0;
          IF LT JUMP posval;      { determine sign }

negval:   AR=PASS AF;
          AR=AR+AF;               { shift left one bit }
          AY0=33;
          AR=AR+AY0;              { add segment offset }
          SR=ASHIFT AR (LO);      { position bits }
          AR=AY0-SR0;             { remove segment offset }
          RTS;
```

(listing continues on next page)

```
posval:   AF=PASS AR;
          AR=AR+AF;                 { shift left one bit }
          AY0=33;
          AR=AR+AY0;                { add segment offset }
          SR=ASHIFT AR (LO);
          AR=SR0-AY0;               { remove segment offset }
          RTS;

.ENDMOD;
```

Listing 11.2 μ-Law Decoder

11.3 PULSE CODE MODULATION USING A-LAW

A-law PCM uses the same approach as μ-law PCM in approximating the logarithmic curve using eight line segments. In A-law conversion, however, the segment endpoints are at even powers of two, rather than offset by 33. On output, only the even bits of the encoded number are inverted in A-law. Otherwise, the conversion is the same. The format of the 8-bit A-law PCM word is the same as the μ-law format; the most significant bit (MSB) is the sign bit, the next three bits contain the segment number, and the last four bits indicate the position within the segment.

11.3.1 A-Law PCM Encoder

The ADSP-2100 A-law encoder, shown in Listing 11.3, is based on the CCITT recommendation G.711. The only deviation from G.711 is the input format. The encoder accepts its input in full fractional format (1.15), instead of the integer format specified G.711, because the ADSP-2100 is optimized for this numeric format. To use this routine with integer input values, you would shift the input three bits to the left (A-law requires a 3-bit shift because it is normalized to 13 bits), maintaining the sign, before calling the routine.

The zero segment values in A-law companding are computed in a slightly different fashion than those of other segments. If the input is less than 128, the segment value is simply the input downshifted by four bits. A test at the beginning of the *a_compress* routine determines whether the input is less than 128 and the routine branches accordingly.

For values in all other segments, the routine determines the exponent of the absolute value of the input using the EXP instruction. The segment value is the value in SE plus seven. After normalization, the most significant non-sign bit is removed by exclusive-ORing with H#4000. The four bits of the segment are positioned by shifting to the right ten bits. The sign and segment bits are determined in the same way as with μ-law encoding.

The final step of the compression process is to invert the even bits of the output. This is accomplished by exclusive-ORing the output with H#55.

```
.MODULE    a_law_compression;

{          This routine determines the 8-bit A-law value from the
           16-bit (left-justified) linear input.

              Calling Parameters
                 AR=16-bit (left-justified) linear input

              Return Values
                 AR=8-bit log (A-law) value with even bits
inverted

             Altered Registers
                AR, AF, AXO, AY0, SR, SE

             Computation Time
                19 cycles

}

.ENTRY       a_compress;

a_compress:  AR=ABS AR;                {Take absolute value}
             AY0=127;                  {Check for zero segment}
             AF=AR-AY0;
             IF GT JUMP upper_seg;
             SR=LSHIFT AR BY -4 (LO);
             AR=H#4000;
             IF NEG AR=PASS 0;
             SR=SR OR LSHIFT AR BY -7 (LO);
             AY0=H#55;
             AR=SR0 XOR AY0;
```

(listing continues on next page)

```
             RTS;
upper_seg:   SE=EXP AR (HI);            {Find exponent
adjustment}
             AX0=SE, SR=NORM AR (LO);   {Normalize input}
             AY0=H#4000;
             AR=SR0 XOR AY0;        {Remove first significant
bit}
             SR=LSHIFT AR BY -10 (LO);  {Shift position bits}
             AR=PASS AY0;
             IF NEG AR=PASS 0;          {Create sign bit}
             SR=SR OR LSHIFT AR BY -7 (LO); {Position sign bit}
             AY0=7;
             AR=AX0+AY0;                {Compute segment}
             IF LT AR=PASS 0;
             SR=SR OR LSHIFT AR BY 4 (LO);{Position segment bits}
             AY0=H#55;
             AR=SR0 XOR AY0;            {Invert bits}
             RTS;
.ENDMOD;
```

Listing 11.3 A-Law Encoder

11.3.2 A-Law PCM Decoder

The *a_expand* routine decodes an A-law PCM value, requiring 18 cycles to produce a result. As in the encoder, the only deviation from the standard is the format of the linear output data, which is 1.15 format rather than the integer format as described in G.711. If integer outputs are required, arithmetically shift the decoder output two bits to the right or remove the shift-by-3 instruction at the end of each routine.

The *a_expand* routine, shown in Listing 11.4, ORs the shifted input value with H#00080800 in the SR register (32-bit register). This simultaneously sets the LSB of the interval and sets the sign bit (takes the absolute value of) the input code word. Also, the 12-bit shift of the input places the segment value in SR1, and the interval in SR0. This value is used to shift the position bits to their proper location.

The sign of the input is determined by adding H#FF80 to it. If the input is negative, the result is greater than zero, and the linear value is negated. In some cases it is necessary to add the interval MSB (32) that was removed during compression. Either 32 or 0 is stored in AF and ORed with the interval bits.

```
.MODULE              A_Law_Expansion;
{
              This routine determines the 16-bit (left-justified)
              linear value from an 8-bit log (a-law) input


              Calling Parameters
                     AR = 8-bit log (a-law) value

              Return Values
                     AR = 16-bit linear output

              Altered Registers
                     AR, AF, AX0, AY0,
                     SR, SE

              Cycle Count
                     19 Cycles

}

.ENTRY               a_expand;

a_expand:            AY0=H#0055;          {Set mask for inversion}
                     AR=AR XOR AY0;       {Even bit inversion}
                     SR1=H#0008;          {Always set sign bit}
                     SR0=H#0800;          {Set LSB of interval}
                     SR=SR OR LSHIFT AR
                     BY 12 (LO);          {Isolate segment,
                                          interval}
                     AY0=32;
                     AX0=AR, AF=PASS AY0;
                     AY0=9;               {Segment bias}
                     AR=SR1-AY0;          {Determine shift value}
                     IF LT AF=PASS 0;     {No extra MSB bit}
                     IF EQ AR=PASS 0;     {No less then zero bits}
```

(listing continues on next page)

```
                    SR=LSHIFT SR0 BY -11 (LO);{Isolate Interval}
                    SE=AR, AR=SR0 OR AF; {Add bit if necessary}
                    SR=LSHIFT AR (LO);   {Position output}
                    SR=LSHIFT SR0 BY 3 (LO);
                     AY0=H#FF80;
                     AR=SR0, AF=AX0+AY0;       {Is sign bit set?}
                     IF LT AR=-SR0;            {Yes, invert word}
                     RTS;

.ENDMOD;
```

Listing 11.4 A-Law Decoder