ADPCM ON TENSILICA

Xiaoling Xu and Fan Mo
Instructor: Professor K.Keutzer and Professor J.Rabaey, TA: S.Weber
EECS, UC Berkeley
May 8, 2000

Abstract: In this project we implement ADPCM (Adaptive Differential Pulse Code Modulation) algorithm on TENSILICA's re-configurable processor. The classical ADPCM algorithm is basically a control-flow style procedure, so that instruction-level parallelism is hard to explore. However, for modern processors, it's possible to process more than one input stream at a time by using SIMD (single instruction multiple data). A modified algorithm is thus devised to allow two encoding or decoding data stream to be packed in a single instruction flow. The unique feature of new instruction generation built in Tensilica is employed to make possible the implementation of the dual-stream ADPCM algorithm. The experimental results show that the new algorithm can provide 4X improvement in encoding rate and 4X improvement in decoding rate, which well exceeds the classical algorithm implemented on the same processor.

I. INTRODUCTION

The ADPCM algorithm is widely used in voice data compression. Each input data sample is a 16-bit signed integer. The encoding procedure compresses it into a 4-bit code. Two successive samples' codes are assembled in one byte. The decoding part takes the the byte codes as input and outputs decompressed 16-bit data samples.
The ADPCM is a differential coding scheme in which each code approximates the difference between the present input value and the previous one. The weighting of the magnitude portion of the difference is adaptive (non-linear). That is, it can change after each sample.

The implementation of ADPCM on microprocessor is quite easy. An overwhelming number of applications use the classical successive-comparison and clamping approach to implement the coding and decoding processes. Constant tables are also required to store step sizes and index incremental values. Such an algorithmic flow doesn't involve multiplication, hence it's easy to program on cheap and simple processor like most micro-controllers. But it's amazing that even for those powerful and omnipotent processors like INTEL X86 series, the same algorithm is used, neglecting plenty of available computational resources. More ironic thing is that still the same algorithm is supposed to be applied to re-configurable architecture like Tensilica until new algorithmic approaches are studied and realized in this project.

The main intuition of our project is got from the unique feature of Tensilica processor, i.e., the user defined new instructions. This allows great flexibility in programming because programmer now has the power of creating new instruction to achieve remarkable performance which is usually impossible in conventional pure software programming. In fact such extra power can be thought of as a hardware-software co-programming facility which provides the users with the ability of deciding whether a certain job is done in software or hardware. Our new algorithm takes advantage of new instruction construction and tries to reach a balance point where the performance is greatly enhanced and hardware overhead is kept as low as possible.

The goal is to implement multi-stream encoding and decoding. The naïve parallel SIMD approach cannot provide expected performance enhancement, because the classical algorithm is entirely control-flow based procedure. So those processors with multiple operating units may not fit this application well, or at least cannot give satisfactory performance even in a two-stream case. A novel way of thinking of this problem is to find a way of encapsulating the multi-stream in a single word and operate on this word. As mentioned earlier, one ADPCM stream requires only 16-bit width, so it's natural to come up with the idea of packing two streams in one 32-bit-wide word. Our new algortihm is mainly based on this idea. However questions like multi-control-flow decoupling and multi-table-lookup remain to be answered. The new instruction feature in Tensilica thus finds its place to show its power.

The rest of this paper is organized as follows. In section II, the classical ADPCM algorithm is revised and the new algorithm which can run two data stream simultaneously is described. In section III, TIE, the user defined instruction feature in Tensilica, is introduced to show how special instructions can be created to expedite the execution of specific program segment. The experimental results are shown in section IV. Three designs based on the modified ADPCM algorithm have been experimeted. Section V is the conclusion.

II. ALGORITHM

2.1 The classical ADPCM algorithm

Figure 1 shows a block diagram of the ADPCM encoding process. A linear input sample X(n) is compared to the previous estimate of that input X(n-l). The difference, d(n), along with the present step size, ss(n), are presented to the encoder logic. This logic, described below, produces an ADPCM output sample. This output sample is also used to update the step size calculation ss(n+l), and is presented to the decoder to compute the linear estimate of the input sample. The encoder accepts the differential value, d(n), from the comparator and the step size, and calculates a 4-bit ADPCM code. The following is a representation of this calculation in pseudocode:

```
Encoding(*input,*output) {
loop(number of samples) {
   X=*input++;
   D=X-X^-1;
   S=StepsizeTable(Index);
   Da=|D|; Code=0; A=0;
   if (Da>S) { Code[2]=1; Da-=S; A+=S; }
   S/=2;
   if (Da>S) { Code[1]=1; Da-=S; A+=S;  }
   S/=2;
   if (Da>S) { Code[0]=1; Da-=S; A+=S;  }
   Code[3]|=(D>0)?0:1;
   X+=(D>0)?A:(-A);
   if (X>32767) X =32767;
   if (X<-32768) X =-32768;
   Index+=IndexTable(Code);
   if (Index>88) Index=88;
   if (Index<0) Index=0;
   X^-1=X;
   *output++=Code;
} }
```

Figure 2 shows a block diagram of the ADPCM decoding process. An ADPCM sample is presented to the decoder. The decoder computes the difference between the previous linear output estimate and the anticipated one. This difference is added to the previous estimate to produce the linear output estimate. The input ADPCM sample is also presented to the step size calculator to compute the step size estimate. The decoder accepts ADPCM code values, L(n), and step size values. It calculates a reproduced differential value, and accumulates an estimated waveform value, X. Here is a pseudocode algorithm:

```
Decoding(*Code,*output) {
   C=*Code++;
   S=StepsizeTable(Index);
   A=0;
   if (C[2]==1) A+=S; S/=2;
   if (C[1]==1) A+=S; S/=2;
   if (C[0]==1) A+=S;
   if (Code[3]==1) X=X^-1-A;
   else  X=X^-1+A;
```

```
    if (X>32767) X =32767;
    if (X<-32768) X =-32768;
    Index+=IndexTable(Code);
    if (Index>88) Index=88;
    if (Index<0) Index=0;
    *output++=X;
    X^-1=X;
}
```

For both the encoding and decoding process, the ADPCM algorithm adjusts the quantizer step size based on the most recent ADPCM value. The step size for the next sample, n+l, is calculated with a two-stage lookup table. First the magnitude of the ADPCM code is used as an index to look up an adjustment factor as shown in indexTable. Then that adjustment factor is used to move an index pointer in stepsizeTable.

```
indexTable[16] = {
   -1, -1, -1, -1, 2, 4, 6, 8,
   -1, -1, -1, -1, 2, 4, 6, 8
};
stepsizeTable[89] = {
   7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
   19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
   50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
   130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
   337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
   876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
   2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
   5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
   15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767
};
```

The index pointer then points to the new step size. Values greater than 3 will increase the step size. Values less than 4 decrease the step size. This method of adapting the scale factor with changes in the waveform is optimized for voice signals, not square waves or other non-sinusoidal waveforms.

2.2 Alternative approaches to speed up the ADPCM algorithm

To make possible a parallel operation for two stream, no matter hardware level parallelism or instruction level, an urgent problem is to decouple the algorithmic flow from if-then-else structure. The reason is quite simple, for if two parallel if-then-else is executed, it's highly probably that one stream may stall to wait until the other stream finishes. Generally such multi-stream scheme is low efficient, and the problem becomes saturate, or falls into worst case, as the number of stream increases. The reason is that, the time taken in a parallel if-then-else depends on the slowest stream, and when stream number is large, most probably at least one stream will come across its worst case.

In this sub-section, the alternative approaches are seeked in order to get out of the senario of the if-then-else structure.

2.2.1 Multiplication

Since the three succesive comparison-actions in the classical algorithm are actually a division operation. The resultant code can be re-written as:

```
Code[2:0]=Da/S*4; A=Code[2:0]*S;
```

or

```
Code[2:0]=Da*(4/S); A=Code[2:0]*S;
```

Therefore the value of (4/S) can be pre-generated and stored in a table. This approach is attractive because multiplier is usually a built-in component in most modern processor. There's no reason neglecting its existence. By using multiplier, we can make the original control-oriented program more data-oriented, so that ILP is more easily to be explored.

However, as we implemented the idea using Tensilica's processor, the result is not acceptable. Due to the usage of the multiplier, the gate account increased by about 30%, but the performance is not increased at all. Why?

First, the execution of a multiplication instruction may take more than ten clock cycles which is about the same or even more than successive comparison-action operations. If fast multiplier is available(like some DSP), the multiplication method can show its advantage.

Second, also more significantly, memory access becomes the bottle neck in this implementation. Although the added table is not big, i.e., the same size as the step size table which has 89 elements, the table lookup process may take longer time than those operations in-between registers. In addition, more intermediate storage is required, so more memory-register swap instructions may be inserted, and this also causes severe Dcache miss.

Another problem is that our original purpose is to process two input streams at the same time. One 16-bit multiplication will create a 32-bit data. If we want to process two 16-bit multiplication, the result will be 64-bit long, which no longer fit in one register. More overhead is introduced to the dual-stream algorithm.

2.2.2 More tables

Another approach is to use more tables to store all information for all possible branches. The successive comparison-action can be modeled as a binary decision tree. The step size parameter used in each decision node is determined by the current path. So all possible step size for all paths are stored in table and fetched when the certain decision node is reached. The corresponding pseudo code now looks like:

```
Da-=S;
Code[2]=~MSB(Da);
Da-=S'[Code];
Code[1]=~MSB(Da);
Da-=S''[Code];
Code[0]=~MSB(Da);
```

in which MSB() takes the MSB of the variable, and S' and S'' store possible step sizes used in respective decision nodes. It's obvious that the index of S' is the partial code generated in the previous step. Same for S''. The first step will result in two possible partial code, "0--" and "1--" with respect to the sign of the subtraction operation. If "1--" which means that the result of the subtract is positive (notice the complement operation when generating code), thus the next step size S' is S/2. If "0--" which means the previous subtraction is overkill, the next step size should compensate it and becomes –S+S/2. This leads to the formulation of:

```
Sn+1=(partial_code_bit==1)?(Sn/2):(-Sn+Sn/2)
```

In this way the successive if-then-else structure is replaced by successive table lookup operation. No multiplication is required in the code generation part, although multiplication can still be used to calculate A. One drawback of this method is that the table lookup time may become a bottle neck. A more subtle point is how to implement parallel table lookup in multi-stream case. A straightforward solution is to build a big table combining the indices of the multiple tables. However such combination actually creates a "product" of the tables which will become impractical when stream number exceeds two. Another solution is to interleave the table lookup and other operations between different streams, such that variable streams share the same single table in a time-division fashion. Unfortunately this limits the application to hardware-level parallelism.

2.3 Dual-stream

We consider how to parallel two stream in a signle instruction flow. Suppose we assign the lower 16 bits for one stream and the higher 16 bits for another. The key problem is how to isolate the two data streams from each other. However the existent instruction set doesn't satisfy such requirement, for all 32-bit arithmetic or logic operations will all regard the 32-bit word as a whole. Thus the result in the lower part of the word may affect the higher part. Vice versa. An example is

addition. The lower part (0 to 15th bit) carry out will be added to the LSB of the higher part (16th to 31th bit). Therefore, a specific kind of instruction is needed to conduct the operation for lower and higher 16 bits separately. In our algorithm two frequently used operations are addition and clamping. Both of them can be, and have to be modified to obey the "separate" rule.

With such idea in mind, we can set out to solve the problem of control-flow decoupling. First such kind of control-flow style steps are the succesive comparison-actions. One bit of the code is generated according to the comparison result. Comparison is basically a subtraction except the reuslt is not written back. In this case, it's a greater-than comparison. The positive reuslt shows "true greater-than" and negative reuslt shows "false". To decouple if-then structure, we employ the binary decision tree method described above. At each step, just do an addition/subtraction according to the code bit generated at the previous step. The result is then used to generate the code bit for the current step. These two new operations are denoted by new instruction SepAddClamp() and CodeBitGen() respectively. Of course they are both dual-stream operations. Following the dual-stream ADPCM encoding algorithm:

```
dual_stream_Encoding(*inputA,*inputB,*outp) {
loop(number of samples/2) {
    //encapsulate two input 16-bit data in a 32-bit word X;
    X=Two16ToOne32(*inputA++, *inputB++);
    d=0;
    step=Ativ(index, 0);
    X=SepAddClamp(X,step,0xffff);
    c|=CodeBitGen(X); d=SepInc(d,step,c); step=SepShift(step);
    c<<=3;
    X=SepAddClamp(X,step,c);
    c>>=2;
    c|=CodeBitGen(X); d=SepInc(d,step,c); step=SepShift(step);
    c<<=3;
    X=SepAddClamp(X,step,c);
    c>>=2;
    c|=CodeBitGen(X); d=SepInc(d,step,c); step=SepShift(step);
    d=SepInc(d,step,0xffff);
    valpred=SepAddClamp(valpred, d, c);
    c>>=2;
    indexinc=Ativ(c,1);
    index=Addt(index, indexinc);
    index=SepCmpAssLe(index, 0);
    index=SepCmpAssGt(index, 88);
    outputbuffer=c<<4;
    //same as above except the last sentence.
    ….
    *outp++ = outputbuffer&0xff;
    *outp++ = outputbuffer>>8;
}}
```

Notice that two consecutive rounds are unwinded in the above codes. At the expense of increasing code size, the necessity of recording the which part of the code is being assembled (higher 4 bits or lower 4 bits) is then removed. The following codes are the dual-stream decoding part. Similar unwinding is adopted.

```
dual_stream_Decoding(*inp,*outpA, *outpB) {
loop(number of samples/2) {
    inputbuffer=*inp++;
    c=(inputbuffer>>4)&0x0f0f;
    step=Ativ(index, 0);
    indexinc=Ativ(c, 1);
    d=0;
    d=SepInc(d, step, c); step=SepShift(step);
    c<<=1;
    d=SepInc(d, step, c); step=SepShift(step);
    c<<=1;
    d=SepInc(d, step, c); step=SepShift(step);
    d=SepInc(d, step, 0xffff);
    index=Addt(index, indexinc);
    index=SepCmpAssLe(index, 0);
    index=SepCmpAssGt(index, 88);
    valpred=SepAddClamp(valpred, d, c);
    *outp++ = valpred;
    //same as above, except the first sentence.
    c=inputbuffer&0x0f0f;
    …..
}}
```

In the next section, we will discuss how to use tensilica's instruction extension to achieve performance inprovment.

III. IMPLENENTAION USING TENSILICA'S INSTRUCTION EXTENSION

There are several hot spots we found when we did profiling using Tensilica's tools.

1. A big chunk of time is spent on table lookup.
2. Some significant time is spent on doing clamp operations.

3.1 Constant Table Lookup

The original ADPCM algorithm used two tables. Even if the tables are all located in the cache, (which is not always the case), getting data from the table still caused big overhead. The immediately effect is that, for each data access, we need a separate instruction to access memory. By using tensilica's profiling tool, we found that, for the original encoder algorithm, 10% time is spent on table lookup instructions. The other side effect of this is, we need registers to store the data we got from the cache. So the contents of the table will compete with other data for the registers, and this will cause more register/memory swapping instructions.

Fortunately, Tensilica has a special feature: constant table. Constant table is a special memory space comes with the processor, and Tensilica allows the customers to configure this special memory space. Since the memory comes with the processor, accessing the constant table is similar to accessing register. No separate instructions are necessary to do the data access. In addition, there is no need to store the data in the register.

We create a new tie instruction, **Ativ** , to do the table lookup.

3.2 Dual-Clamp Operations

There are a lot of if-else instructions exists in the original algorithm, most of them are clamp operations, i.e. if(a>b) a=b. In our dual-stream algorithm, we desire one instruction to do clamp operation on two data. I.e. we desire some instruction which can implement the following things:

```
if(A[31:16] > B[31:16] && A[15:0] > B[15:0] ) A = B
if(A[31:16] > B[31:16] && A[15:0] < B[15:0] ) A = B[31:16]•A[15:0]
if(A[31:16] < B[31:16] && A[15:0] > B[15:0] ) A = A[32:16]•B[15:0]
if(A[31:16] < B[31:16] && A[15:0] < B[15:0] ) A = A;
```

We used TIE to implement this idea, i.e. **SepCmpAssLe**(A, B) and **SepCmpAssGt**(A, B).

3.3 Dual-Add-Clamp Operations

The major obstacle really confused us is how to put two 17-bit data into one 32-bit register. Observing the original algorithm, we found that there are cases where some add operations will cause the internal data be 17-bit long. When we do the dual-stream operations, since we will put two data into one register and do operations on the two data in the same time, we have problems when the two data become 17 bit long each. However, we also found that the algorithm doesn't really "want" the data be such big: they will clamp the data into 16 bit long anyway.

There are no places where the 17-bit long data is needed. The flow is like this:

```
….
a = a + diff;   //where a may be 17-bit long.
if (a>2^15-1) a=2^15-1;
….
```

What we decide to do is combine these two instructions together to generate a new instruction: **SepAddClamp**. This will avoid the 17-bit long data to be generated.

A complete .tie description file can be found in Appendix A.

IV. EXPERIMENTAL RESULTS

We have implemented the dual-stream adpcm algorithm in 3 different ways:

Design 1: Using Tensilica's instruction set architecture with no TIE and constant table lookup

Design 2: Using Tensilica's instruction set architecture with new instructions

Design 3: Using Tensilica's instruction set architecture with new instructions and constant table lookup

We tested our designs using 3 different waveform: sin/linear/random, in which each has three different step size: min/typical/maximum. We found that, by using Tensilica's instruction extension, we improved the performance for both encoder and decoder by 2X. With the usage of constant tables, we improved performance of both encoder and decoder by another 2X. In addition, the hardware overhead for these new instructions are not very larger, all the tie instructions need is 2 16-bit adder, plus some 2-to-1 multiplex. Table 1, 2 and 3 list a summary. The compasiron of different algorithms implemented on Tensilica is shown if Figure 3. In addition, we compared our implementation on Tensilica with other implementations using different processor, which can be found in Table 4.

Table 1: Encoder Instructions Analysis using 10000 data samples

	Design 1	Design 2	Design 3
# of instructions	976749	705456	576152
# of branch instructions	240011	25010	25010
# of memory access instructions	210051	120130	40309
# of Dcache miss instructions	84656	54909	12768

Table 2: Decoder Instructions Analysis using 10000 data samples

	Design 1	Design 2	Design 3
# of instructions	815621	420115	225119
# of branch instructions	198858	5010	5010
# of memory access instructions	190052	120042	11037
# of Dcache miss instructions	26989	14039	10798

Table 3: performance and cost analysis using 10000 data samples

	Design 1	Design 2	Design 3
clock rate (MHz)	141	141	141
gate estimation	50500	51500	51500
encoder/decoder rate(Mbit/sec)	1.26/2.12	2.26/5.87	4.23/12.1

Table 4: Comparison

Processor	Encoding	Decoding
R4000 Indigo	1.1M	1.7M
R3000 Indigo	410K	850K
Sun SLC	250K	420K
Mac-Iisi	21K	35K
486/DX2-33 SCO	550K	865K
486/33 linux	278K	464K
386/33 gcc	117K	168K
TMS320C5000 *	40K	40K
Tensilica/141	**4.23M**	**12.1M**

*this result is achieved with optimization emphasis on low power.

V. CONCLUSION

In this project, we create a dual-stream ADPCM algorithm which is specific for Tensilica re-configurable architecture. The TIE new instruction feature is heavily relied on so that the performance of the algorithm is enhanced significantly. The experimental results show that our new algorithm can achieve a 4X encoding/decoding rate improvement. Many valuable characteristics like cache hit/miss analysis and multiplication and table lookup performance are well studied.

Future work may include implementing the algorithm using Texas Instrument's C54 DSP processor. In addition, there is possiblity to implement a 4-stream adpcm algorithm by using Tensilica TIE features.

REFERENCES

[1] "Dialogic ADPCM Algorithm", Dialogic Corporation, 1988

[2] http://www.tensilica.com/

APPENDIX A: TIE INSTRUCTION

```
// Instructions description:
// Addt ->
// out[high] = i[high] + j[high]
// out[low] = i[low] + j[low]
// SepCmpAssLe ->
// if(i<constant) i=constant
// SepCmpAssGt ->
// if(i>constant) i=constant
// SepShift ->
// a[low] = a[low] >> 1
// a[high] = a[high] >> 1
// SepAddClamp ->
// a[low] = a[low] +/- b[low] (depends on art value)
// if a[low] > 32767 a[low] = 32767
// if a[low] < -32768 a[low] = -32768
// SepInc ->
// if (c) a[low] = a[low] + b[low] else a[low] =
a[low]
// CodeBitGen
// if ((a&0x80000000)==0) c=0x0400;
// if ((a&0x00008000)==0) c|=0x0004;
// Ativ ->
// table lookup

opcode Addt  op2=4'b0000   CUST0
iclass New0Instructions {
  Addt
} {
  out arr, in ars, in art
}
semantic New0Instructions {
  Addt
} {
  //Addt
  wire [15:0] low;
  wire [15:0] high;
  assign low = ars[15:0] + art[15:0];
  assign high = ars[31:16] + art[31:16];
  assign arr = {high, low};
}

opcode SepCmpAssLe op2=4'b0001 CUST0
field com_field Inst[11:4]
operand com com_field {
  {8{com_field[7]}, com_field}
}
    {com[7:0]}
```

```
iclass New1Instructions {
  SepCmpAssLe
} {
  inout arr, in com
}
semantic New1Instructions {
  SepCmpAssLe
} {

  //SepCmpAssLe
  wire [15:0] tmp1;
  wire [15:0] tmp2;
  wire [15:0] tmp_low;
  wire [15:0] tmp_high;
  assign tmp1 = arr[15:0] - com[15:0];
  assign tmp_low = tmp1[15] ? com[15:0] : arr[15:0];
  assign tmp2 = arr[31:16] - com[15:0];
  assign tmp_high = tmp2[15] ? com[15:0] :
arr[31:16];
  assign arr = {tmp_high, tmp_low};
}

opcode SepCmpAssGt op2=4'b0010 CUST0
iclass New2Instructions {
  SepCmpAssGt
} {
  inout arr, in com
}
semantic New2Instructions {
  SepCmpAssGt
} {

  //SepCmpAssLe
  wire [15:0] tmp1;
  wire [15:0] tmp2;
  wire [15:0] tmp_low;
  wire [15:0] tmp_high;
  assign tmp1 = arr[15:0] - com[15:0];
  assign tmp_low = tmp1[15] ? arr[15:0] : com[15:0];
  assign tmp2 = arr[31:16] - com[15:0];
  assign tmp_high = tmp2[15] ? arr[15:0] :
com[31:16];
  assign arr = {tmp_high, tmp_low};
}

opcode SepShift op2=4'b0011 CUST0
iclass New3Instructions {
  SepShift
} {
  inout arr
}
semantic New3Instructions {
  SepShift
} {
  wire [15:0] tmp0;
  wire [15:0] tmp1;
  assign tmp0 = {arr[15], arr[15:1]};
  assign tmp1 = {arr[31], arr[31:16]};
  assign arr = {tmp1, tmp0};
}

opcode SepAddClamp op2=4'b0100 CUST0
iclass New4Instructions {
  SepAddClamp
} {
  inout arr, in ars, in art
}
semantic New4Instructions {
  SepAddClamp
} {
  wire [16:0] aL;
  wire [16:0] aH;
  wire [16:0] bL;
  wire [16:0] bH;
  wire [16:0] tmpLM;
  wire [16:0] tmpLP;
  wire [16:0] tmpHM;
  wire [16:0] tmpHP;
  wire ofL1;
  wire ofL2;
  wire ofH1;
  wire ofH2;
  wire [15:0] tmpLMF;
  wire [15:0] tmpLPF;
  wire [15:0] tmpHMF;
  wire [15:0] tmpHPF;
  wire [15:0] tmpL;
  wire [15:0] tmpH;
  assign aL = {arr[15], arr[15:0]};
  assign aH = {arr[31], arr[31:16]};
  assign bL = {{0}, ars[15:0]};
  assign bH = {{0}, ars[31:16]};
  assign tmpLM = aL - bL;
  assign tmpHM = aH - bH;
  assign tmpLP = aL + bL;
  assign tmpHP = aH + bH;
  assign ofL1 = arr[15] && ((!tmpLM[15]) ||
(!tmpLM[16]));
  assign ofH1 = arr[31] && ((!tmpLM[15]) ||
(!tmpLM[16]));
  assign ofL2 = (!arr[15]) && (tmpLP[15] ||
tmpLP[16]);
  assign ofH2 = (!arr[31]) && (tmpLP[15] ||
tmpLP[16]);
  assign tmpLMF = ofL1 ? -32768 : tmpLM[15:0];
  assign tmpHMF = ofH1 ? -32768 : tmpHM[15:0];
  assign tmpLPF = ofL2 ? 32767 : tmpLP[15:0];
  assign tmpHPF = ofH2 ? 32767 : tmpHP[15:0];
  assign tmpL = art[5] ? tmpLMF : tmpLPF;
  assign tmpH = art[13] ? tmpHMF : tmpHPF;
  assign arr = {tmpH, tmpL};
}

opcode SepInc op2=4'b0101 CUST0
iclass New5Instructions {
  SepInc
} {
  inout arr, in ars, in art
}
semantic New5Instructions {
  SepInc
} {
  wire [15:0] tmpL;
  wire [15:0] tmpH;
  assign tmpL = art[2] ? arr[15:0] + ars[15:0] :
arr[15:0];
  assign tmpH = art[10] ? arr[31:16] + ars[31:16] :
arr[31:16];
  assign arr = {tmpH, tmpL};
}

opcode CodeBitGen op2=4'b0110 CUST0
iclass New6Instructions {
  CodeBitGen
} {
  out arr, in ars
}
semantic New6Instructions {
  CodeBitGen
} {
  wire [15:0] tmp1;
  wire [15:0] tmp2;
  assign tmp1 = ars[31] ? 0 : 1024;
  assign tmp2 = ars[15] ? tmp1 : tmp1+4;
  assign arr = tmp2;
}

opcode Ativ op2=4'b0111 CUST0
table indexTable 16 16 {
```

```
  -1, -1, -1, -1, 2, 4, 6, 8, -1, -1, -1, -1, 2, 4,
6, 8
}
table stepTable 16 89 {
  7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
  19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
  50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
  130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
  337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
  876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878,
2066,
  2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428,
4871, 5358,
  5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487,
12635, 13899,
  15289, 16818, 18500, 20350, 22385, 24623, 27086,
29794, 32767
}
iclass New7Instructions {
  Ativ
} {
  out arr, in ars, in art
}
semantic NewInstructions {
  Ativ
} {
  wire [15:0] index1;
  wire [15:0] index2;
  wire [15:0] tmp1;
  wire [15:0] tmp2;
  // Ativ
  assign index1 = art[0] ? ars[8:0] : ars[15:0];
  assign index2 = art[0] ? ars[15:8] : ars[31:16];
  assign tmp1 = art[0] ? indexTable[index1] :
stepTable[index1];
  assign tmp2 = art[0] ? indexTable[index2] :
stepTable[index2];
  assign arr = {tmp2, tmp1};
}
```

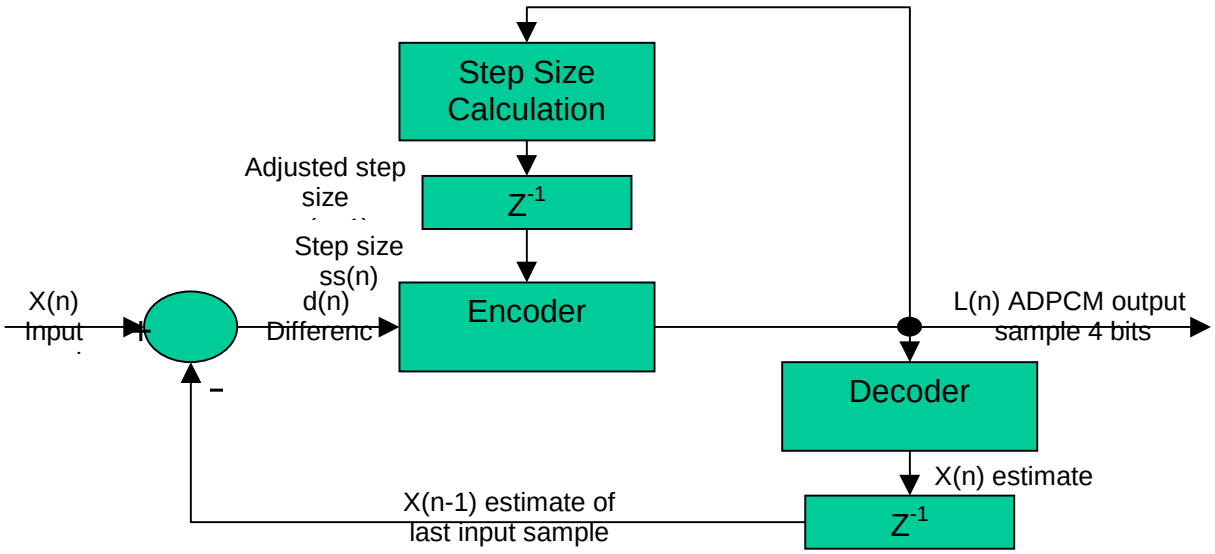


Figure 1. Encoding diagram

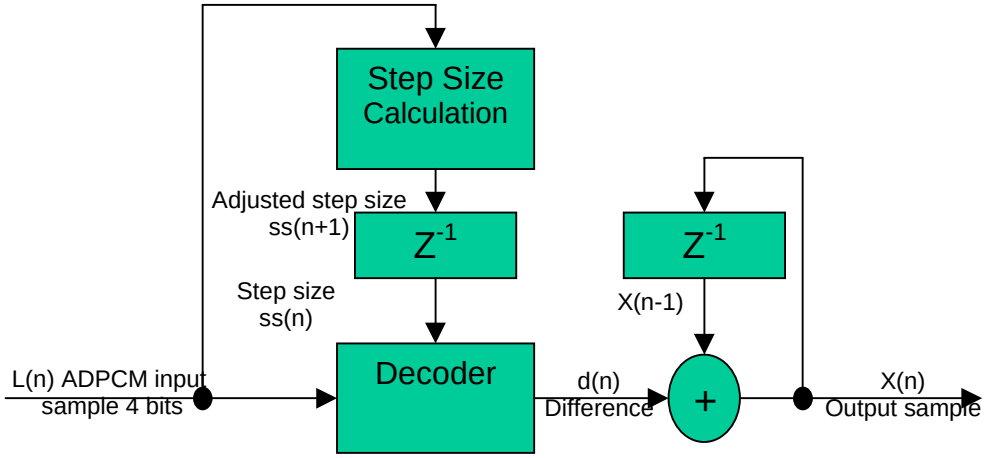
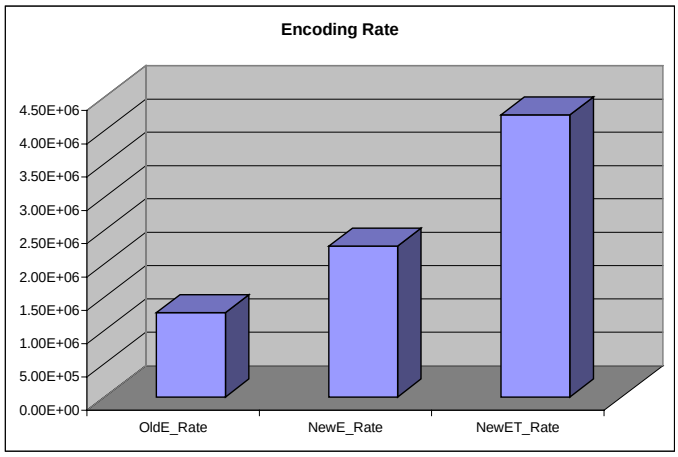


Figure 2. Decoding diagram



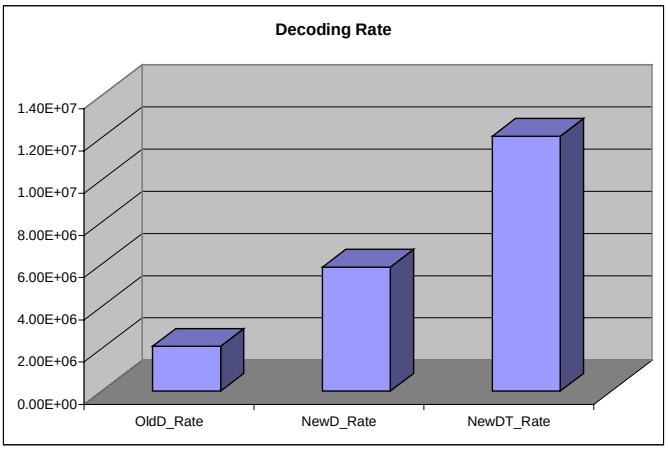


Figure 3 Comparison of different algorithms implemented on Tensilica