Application Report
SPRAA83–August 2005

A Novel Approach For Increasing Voice Channel Density Exploiting VLIW DSP

Manjunatha C. Pawate *Software Development Systems*

ABSTRACT

In this application report, a novel technique is presented for processing multiple voice channels using the standard ITU G.726 codec on a TMS320C6416 digital signal processor (DSP) from Texas Instruments Incorporated. High channel density is a key requirement for central office applications. Previous implementations, process one channel at a time and the number of channels supported by a device is then determined by its processing speed in MHz and memory supported. For example, a 200MHz device supports 20 channels of G.726 if each channel requires 10MHz to process. In this applicaton report, the inherent, multiple processing units are exploited and available on the TMS320C6416 device to simultaneously process two or more channels at a time. As a result, the effective MHz required for each channel is reduced significantly. Using this novel approach shows that for G.726 codec, the effective MHz per channel is reduced from 6.25MHz to 3.22MHz when two channels at a time are processed. This opens the opportunity for processing three or more channels at a time.

Contents

List of Figures

Trademarks

TMS320C64x is a trademark of Texas Instruments.

1 Introduction

This application report discusses how to implement a 2-channel-processing method and list the details. This enables a 200MHz device to support 62 channels instead of 32 channels if it were implemented in a typical 1-channel-processing method.

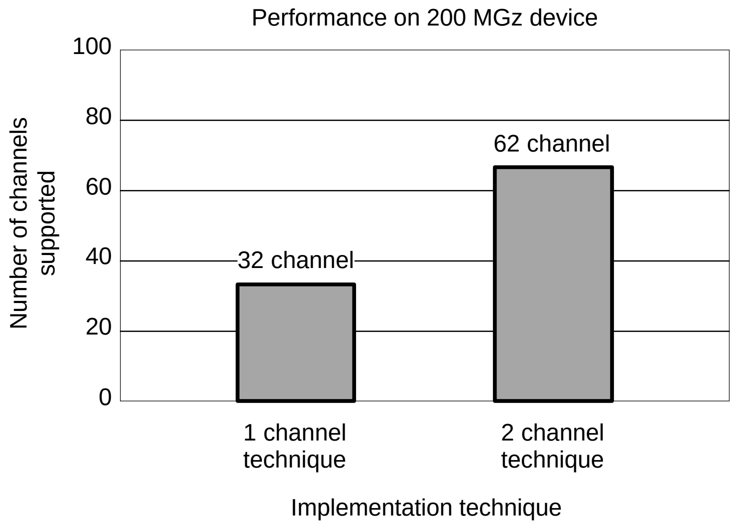


Figure 1. Implementation Technique

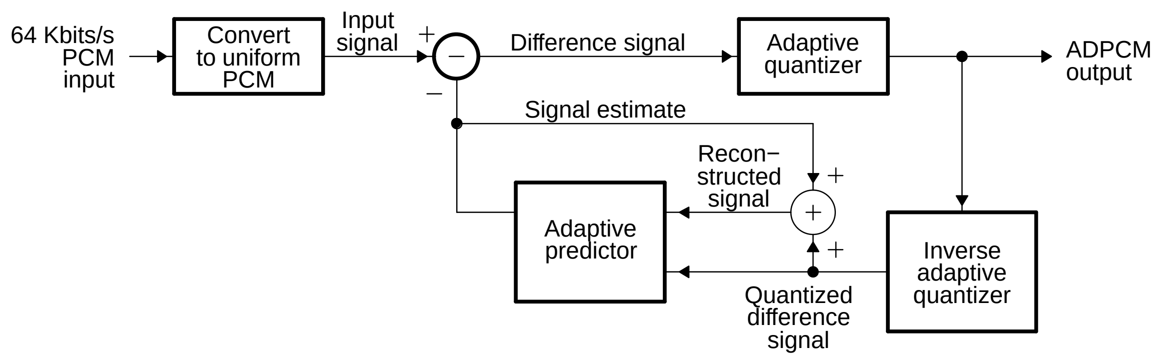
The above graph shows the number of channels supported by one-channel and two-channel implementation respectively. The x-axis represents the implementation technique and y-axis represents the number of channels supported on a 200 MHz processor. From the above graph, it is shown that one-channel implementation supports 32 channels approximately; whereas, two-channel implementation supports 62 channels approximately on the same 200 MHz processor.

Adaptive differential pulse code modulation (ADPCM) is a widely used voice-coding standard for communications links within the worldwide telecommunications network. Within the International Telecommunication Union (ITU), several standards are defined and the most commonly used speech codec is G.726.

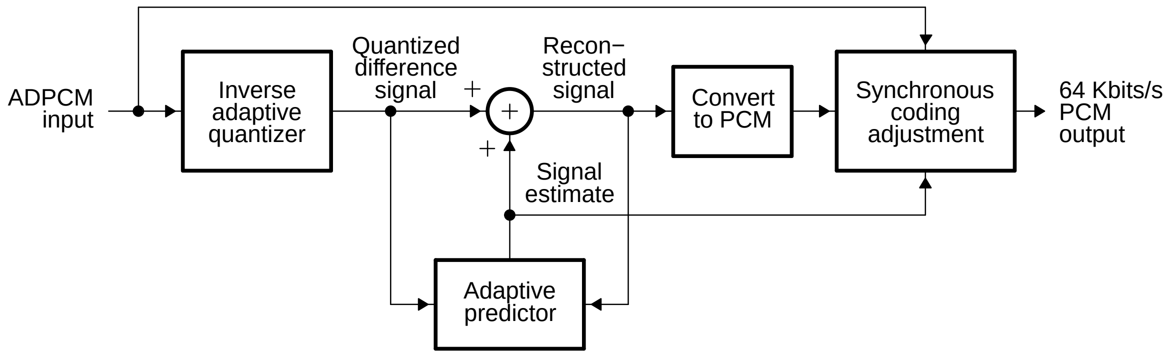
2 What Is G.726 ADPCM?

ADPCM is a voice-compression algorithm that works in the time domain by predicting the next time sample based on the spectrum and amplitude of the previous data. The block diagram shown below shows the principal of an ADPCM voice coder. The adaptive predictor is used to predict the spectral quantities of the next sample. The difference between this prediction and the real signal is then quantized using an adaptive filter, which bases it quantization levels on the past accuracy of recent previous samples. This value is then used to update both the predictor and the quantizer for the next sample.

The decoder works in a similar manner, except that there is an additional stage at the end called synchronous coding adjustment. This mimics the effects of tandeming several voice vocoders in sequence and adjusts the output to reduce the risk of an ADPCM voice coder further down the line to make a different decision as to which ADPCM value to compress.



a) Encoder



b) Decoder

Figure 2. ADPCM Voice Coder

G.726 is a standard ADPCM algorithm specified by the ITU to reduce the 64-KBps A-law or m-law logarithmic data of a normal telephone line to any of 16 Kbps, 24 Kbps, 32 Kbps, or 40 Kbps. The full mathematical specification is ITU copyright and can be found in the G.726 specification published by the ITU. G.721 is an older ITU standard for ADPCM that only supports 32 Kbps. G.721 (1988) is a full bit-compatible G.726.

3 Why C64x?

The TMS320C64x™ digital signal processors (DSPs) architecture is a very long instruction word (VLIW) architecture with multiple execution units. In C64x device, there are eight independent functional units such as: L1, M1, S1, D1, L2, M2, D2, and S2. Each of these units perform independent operations in the same cycle. These functional units are used in parallel to execute up to eight instructions in a single clock cycle. This opens the door for a *n-channel* implementation of any algorithm on this (C64x) device.

4 Implementation Of G.726 On C64x Architecture

One-channel, two-channel implementation of G.726 algorithm have been done on C64x device. N-channel implementation of G.726 requires reading of n input samples from n different files and for each function we have passed a buffer along with functional arguments to pass the inputs to the functions as shown in the figure below. The same buffer is used to collect the outputs of n channels. C64x can return only one value, but two values are required. At the end of each function, the address of buffer and assigning to a pointer variable are returned. Finally, these different n outputs are written to n different files. The output is verified by using ‘bit exact matching’ rule.

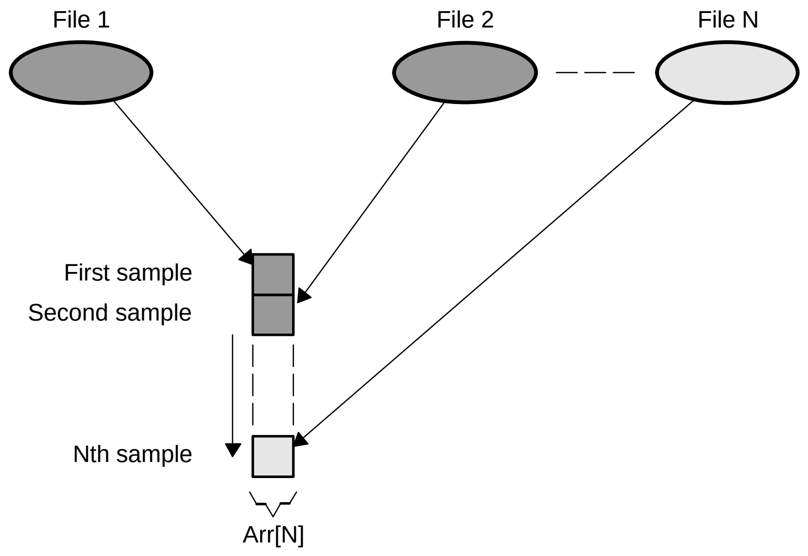


Figure 3. Channel Implementation

In one-channel implementation, all of the functional units cannot be used to execute eight instructions in parallel due to data dependency. Thus, most of the functional units are left unused. In n-channel implementation, most of these functional units can be in parallel. Thus in n-channel implementation we can efficiently utilize the architecture. However, n-channel implementation may leads to consume more CPU clock cycles. This overhead may be due to both architectural constraints and algorithmic constraints. For example, one-channel implementation on C64x consumes 5.6 Mega cycles; two-channel implementation consumes 6.1 Mega cycles (effective consumption of cycles for one-channel is 3.05 Mega cycles); and four-channel implementation consumes 10 Mega cycles (effective consumption of cycles for one-channel is 2.5 Mega cycles).

L1
S1
M1
D1
D2
M2
S2
L2

Figure 4. Functional Units of C64x

Above shown units are the functional units of C64x. Each unit can perform independent operation. So we can execute unto eight instructions in parallel in a single clock cycle as shown in the example below.

For example:

```
      ADD     .L1    A16, A17, A16;
||    ADD     .L2    B18, B17, B18;
||    SHL     .S1    A18, 3, A19;
||    SHR     .S2    B29, 4, B19;
||    MPY     .M1    A22, A24, A23;
||    MPY     .M2    B23, B27, B28;
||    LDH     .D1    *A4, A31;
||    LDDW    .D2    *B4, B31:B30;
```

For example consider a C function,

```
int    reconstruct (
int    sign,               /* 0 for non-negative value */
int    dqln,              /* G.72x codeword */
int    y,                /* Step size multiplier */
{
short    dql;          /* Log of 'dq' magnitude */
short    dex;         /* Integer part of log */
short    dqt;
short    dq;         /* Reconstructed difference signal sample */
dql = dqln + (y >> 2);    /* ADDA */
if (dql < 0)
{
return ((sign) ? -0x8000 : 0);
}
else
{                                       /*ANTILOG */
dex = (dql >> 7) & 15;
        dqt = 128 + (dql & 127);
        dq = (dqt << 7) >> (14 - dex);
        return ((sign)? (dq - 0x8000) : dq);
}
}
```

The above function is a reconstruct module taken from the G.726 algorithm. This function reconstructs the original signal back

4.1 One Channel Implementation Technique

Function has three parameters, the function prototype is not changed, It is as shown below.

Initially the C statement for one channel would look like

```
int   reconstruct (int sign, int  dqln, int  y) ;
```

The assembly code for the 1 channel implementation of the above reconstruct function is as given below.

```
.global _reconstruct;
.text
_reconstruct:
      SHR   .S1 A6, 2, A16             ;Y1>>2
||    ZERO  .L2 B7                     ;cycle
||    MPY   .M2 0,B9, B9               ;
||    MV    .L1 B8, A31                ;address of an array
```

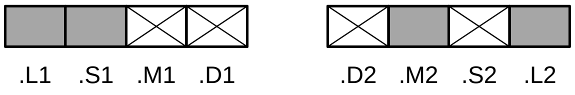


Figure 5. First Cycle

Here in the first cycle we right shift the value of Y1 of the first channel by 2. Address of the array is passed from B8 to A31, and B7 and B9 are set to zero.

```
     ADD   .D1 B4,A16,A16    ;DQL1    1st CH
  || CMPEQ .L1 A4,0,A0       ;SIGN1   1st CH
  || MVK   .S2 128,B17       ;128
  || MVK   .S1 127,A17       ;27 cycle
```

Single clock cycle

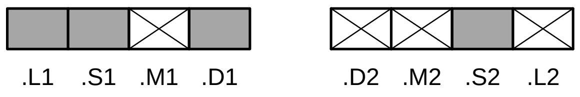


Figure 6. Second Cycle

In the second cycle, the value of the sign1 of the first channel is compared equal to zero and result is kept in A0. Addition of dqln1 and y>>2 is done in the same cycle. Constants 127 and 128 are moved to the A17 and B17 registers.

```
     SHR    .S1 A16,7,A18   ;DQL1 >> 7     1st CH
  || AND    .D1 A16,A17,A19 ;DQL1 & 127    1st CH
  || CMPLT  .L1 A16,0,A1    ;DQL1 < 0      1st CH
  || MVKL   .S2 −0X8000,B20 ;
```

Single clock cycle

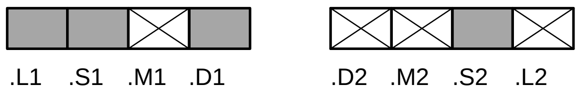


Figure 7. Third Cycle

In the third cycle, the value of DQL1 is compared with 0. If it is less than zero, then the value is returned else it proceeds for the further computations. Comparison of DQL1 is done in the same cycle. ANDing of the DQL1 127 is done in this cycle only and also shifting operations are also done here.

```
     AND  .L1   A18,15,A18    ;DEX1 1st CH
  || ADD  .D1  B17,A19,A19    ;DQT1 1st CH
  || MVKL .S1  0X8000,A20     ;
  || B      .S2  B3           ;RETURN
```

Single clock cycle

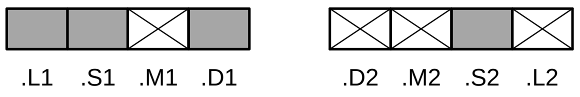


Figure 8. Fourth Cycle

The value of DEX1is computed in this cycle, and DQT1 of the single channel. We also move constant values to the A20 and B20 registers which can be used in next coming cycles. Branch operation is also done in this cycle.

```
   SHL     .S1 A19,7,A19    ;DQT1 << 7           Single
|| SUB     .L1 14,A18,A18   ;14 − DEX1           clock
|| [!A0]MV .D2 B20,B7       ;                    cycle
```

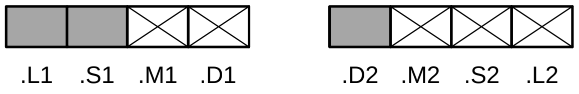


Figure 9. Fifth Cycle

In this cycle, we shift left the value of DQT1 of 1st channel also subtract the DEX1 of 1st channel from constant 14.

```
      SHR .S1 A19, A18, A18    ;DQ1
||    [A1]MV .D2 B7, A4       ;OUTPUT
```

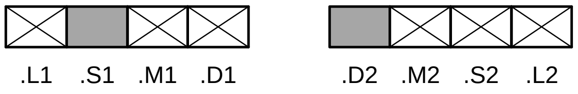


Figure 10. Sixth Cycle

Here we shift right DQ1 of first channel by the value resulted from 14-DEX1 besides storing the first channel output (only if DQL1 < 0) .

```
SUB .S1 A18, A20, A24         ;DQ1-0x8000
```

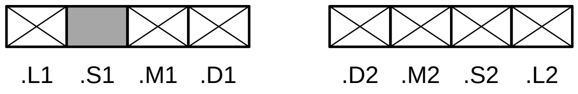


Figure 11. Seventh Cycle

In this clock cycle, values of DQ1-0x8000 are computed.

```
[!A0]MV .D1 A24,A18            ;1st CH Single Clock Cycle
```

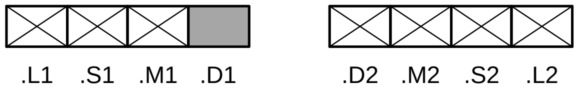


Figure 12. Eighth Cycle

Here content of A24 is moved to A18 if the condition holds good.

```
[!A1]MV .D1  A18,A4    ;     OUTPUT    1st CH Single Clock Cycle
```

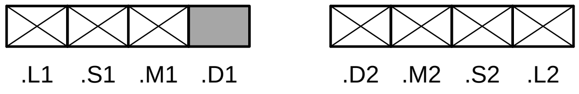


Figure 13. Ninth Cycle

Finally, the output of one channel is stored in the A4 register.

4.2 Two Channel Implementation Technique

For 2-channel implementation on C64x, the same function operations is performed on two different input samples in parallel. There are eight independent functional units on C64x. Those eight can exploit the parallelism efficiently with very negligible overhead in terms of cycles. This negligible overhead is due to either architectural constraints or algorithmic constraints.

C function prototype for two channel implementation is as shown below.

```
int reconstruct(int sign, int dqln, int y, int dqln2, int y2, int *);
```

For two channel implementation, two return values (i.e. one from each channel) are required. C64x can only return one value through A4 register, but the two return values are required. To resolve this, the address of an array is passed to write the required number of return values into the array and address of the array is returned into the A4 register.

The number of arguments passed in the function is also increased because the required values for both first and second channel are passed.

The assembly code for the two channel implementation of the above reconstruct function is as given below.

```
.global _reconstruct      ;
.text
_reconstruct:
   SHR  .S1 A6,2, A16    ;Y1>>2    1st CH
|| SHR  .S2 B6,2, B16    ;Y2>>2    2nd CH
|| LDW  .D2 *+B8[0], B30 ;LD SIGN2 2nd CH
|| ZERO  .L2 B7          ;
|| MPY  .M2 0,B9, B9     ;
|| MV   .L1 B8, A31      ;address of an array
```

Single clock cycle

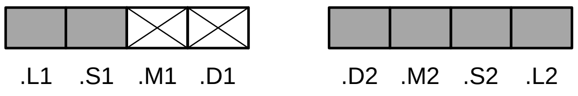


Figure 14. First Cycle

Here in the first cycle we right shift the value of Y1 of the first channel by 2 in parallel with the Y2 of the second channel by 2.Address of the array is passed from B8 to A31, and B7 and B9 are set to zero.

```
   ADD   .D1 B4, A16, A16  ;DQL1    1st CH
|| CMPEQ .L1 A4, 0, A0     ;SIGN1   1st CH
|| ADD   .D2 A8, B16, B16  ;DQL2    2nd CH
|| MVK   .S2 128, B17      ;128
|| MVK   .S1 127, A17      ;127
```

Single clock cycle

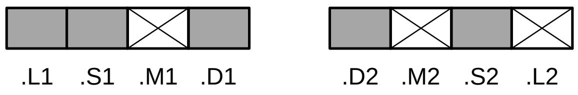


Figure 15. Second Cycle

In the second cycle, the value of the sign1 of the first channel is compared equal to zero and result is kept in A0.In the same cycle we cannot compare the Sign2 because we are loading the value of Sign2, the value of Sign2 will be available later. Addition of dqln1 and y>>2 is done for both the channel in the same cycle. Constants 127 and 128 are moved to the A17 and B17 registers.

```
   SHR    .S1 A16, 7, A18   ;DQL1 >> 7  1st CH
||  AND   .D1 A16, A17, A19 ;DQL1 & 127 1st CH
||  CMPLT .L1 A16, 0, A1    ;DQL1 < 0   1st CH
||  AND   .D2 B16, A17, B19 ;DQL2 & 127 2nd CH
||  SHR   .S2 B16, 7, B18   ;DQL2 >> 7  2nd CH
||  CMPLT .L2 B16, 0, B1    ;DQL2 < 0   2nd CH
```

Single clock cycle

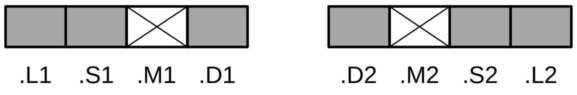


Figure 16. Third Cycle

In the third cycle, we are comparing the value of DQL1 with 0. If it is less than zero, then the value is returned else it proceeds for the further computations. Both the comparisons DQL1 and DQL2 are done in the same cycle. ANDing of the DQL1 and DQL2 with 127 is done in this cycle only and also shifting operations are also done here. In this cycle, out of 8 units we have used only six because we cannot use M units, as there are no multiplication operations present.

```
   AND  .L1 A18,15, A18     ;DEX1     1st CH
|| ADD  .D1 B17, A19, A19   ;DQT1     1st CH
|| AND  .L2 B18,15, B18     ;DEX2     2nd CH
|| ADD  .D2 B17, B19, B19   ;DQT2     2nd CH
|| MVKL .S1 0X8000, A20     ;
|| MVKL .S2 −0X8000, B20    ;
```

Single clock cycle

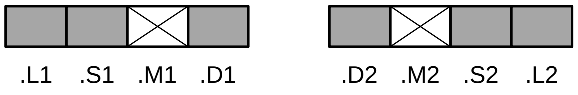


Figure 17. Fourth Cycle

The value of DEX1 and DEX2 is computed in this cycle. DQT1 and DQT2 of both the channels simultaneously. The constant values to the A20 and B20 registers are moved so that they can be used in next coming cycles.

```
B .S2 B3           ;RETURN
```

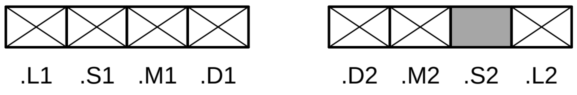


Figure 18. Fifth Cycle

Branch operation is done in this cycle. This operation is only done in this cycle because it cannot be performed in the previous cycle as we have used S2 unit for other purpose.

```
   SHL    .S1 A19,7, A19   ;DQT1 << 7    1st CH
|| SUB    .L1 14,A18, A18  ;14−DEX1      1st CH
|| [!A0]MV .D2 B20, B7     ;             1st CH
|| SHL    .S2 B19,7, B19  ;DQT2 << 7    2nd CH
|| SUB    .L2 14,B18, B18 ;14−DEX2      2nd CH
```

Single clock cycle

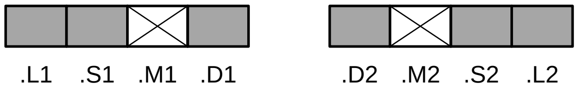


Figure 19. Sixth Cycle

In this cycle, the value of DQT1 of 1st channel and value of DQT2 of 2nd channel is shifted to the left in parallel and also subtract the DEX1 and DEX2 of 1st and 2nd channel respectively from constant 14 simultaneously.

```
    SHR      .S1 A19, A18, A18 ;DQ1       1st CH
||  [A1]STW  .D2 B7,*+B8[2]    ;OUTPUT    1st CH
||  SHR      .S2 B19, B18, B18 ;DQ2       2nd CH
||  CMPEQ    .L2 B30,0, B0    ;SIGN2 = 0 2nd CH
```

Single clock cycle

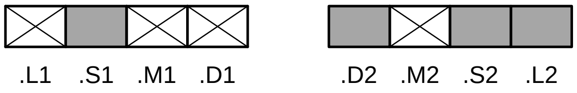


Figure 20. Seventh Cycle

Here we shift right DQ1 of first channel by the value resulted from 14-DEX1 and DQ2 of second by the value of 14-DEX2 in parallel besides storing the first channel output (only if DQL1 < 0) and comparing SIGN2 of second channel with constant 0.

```
   SUB    .S1 A18, A20, A24  ;DQ1−0X8000    1st CH
|| SUB    .S2 B18, A20, B24  ;DQ2−0X8000   2nd CH
|| [!B0]MV .L2 B20, B9        ;             2nd CH
```

Single clock cycle

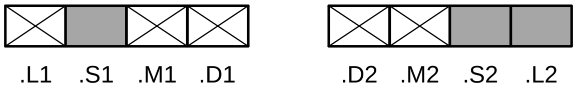


Figure 21. Eighth Cycle

In this clock cycle, values of DQ1-0x8000 and DQ2-0x8000 are computed and contents of B20 register are moved to register B9.

```
   [!A0]MV    .D1 A24, A18    ;       1st CH
|| [!B0]MV    .S2 B24, B18    ;       2nd CH
|| [B1]STW    .D2 B9,*+B8[3]  ;OUTPUT 2nd CH
```

Single clock cycle

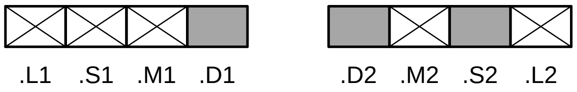


Figure 22. Ninth Cycle

Here, content of A24 and content of B24 are moved to A18 and B18 respectively. The output of second channel is stored into the array (only if DQl2 < 0).

```
   [!A1]STW  .D2 A18,*+B8[2]  ;OUTPUT  1st CH
|| [!B1]STW  .D1 B18,*+A31[3] ;OUTPUT  2nd CH
```

Single clock cycle

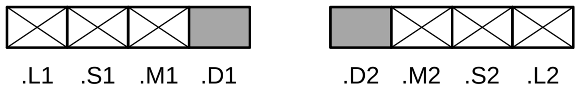


Figure 23. Tenth Cycle

Finally, the output of two different channels is stored in the array.

5 Comparing Cycles Consumed By Reconstruct Function

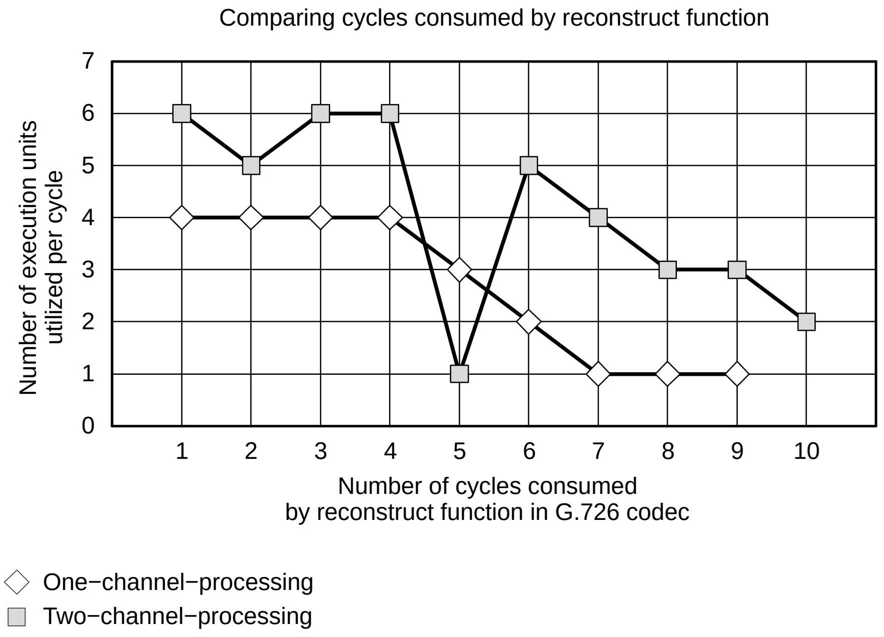


Figure 24. Comparing Cycles

TMS320C64x can execute up to eight 32-bit instructions per cycle. The core C64x CPU has eight independent functional units namely .L1, .M1, .S1, .D1, .D2, .S2, .M2, .L2. These eight functional units contain:

- Two multipliers
- Six ALU's

The above graph shows the functional units active in each cycle and number of cycles taken by 1-channel, 2-channel of reconstruct function of G.726. From the above graph, it is clear that both 1-channel and 2-channel implementations take the same number of cycles but 2-channel implementation utilizes more functional units (i.e., architecture utilization factor (AUF)) are more. Thus, when any application requires more number of channels to be opened, two-channel implementation effectively reduces the consumption of CPU clock cycles.

6 Conclusion

In this application report, a new method is proposed for utilizing the architectural features of a VLIW DSP such as the C6416 processor. The implementation *simultaneously* processes *two channels at a given time*. This significantly reduces the per-channel cycles consumed by a given algorithm. The G.726 speech compression algorithm was taken as an example in the study. We demonstrate a 2x improvement in the channel density using this novel technique with minimal overhead. A 200MHz, C64x device can process 64 channels instead of 32 by using this novel approach.

In the future, other benefits of processing will be explored in three or more channels simultaneously.

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265