

# A Low-Power Reconfigurable Datapath for Advanced Speech Coding Algorithms

Konstantinos Katsoulakis<sup>1</sup>, Tughrul Arslan<sup>1,2</sup>, Tony Kirkham<sup>3</sup>, Sami Khawam<sup>1</sup>

<sup>1</sup>*School of Electronics and Engineering  
The University of Edinburgh  
The Kings Buildings,  
Mayfield Road, Edinburgh,  
EH9 3JL, UK  
[k.katsoulakis@ed.ac.uk](mailto:k.katsoulakis@ed.ac.uk)*

<sup>2</sup>*Institute for System Level Integration  
The Alba Campus, Livingston,  
EH54 7EG, UK  
[tugrul.arslan@ed.ac.uk](mailto:tugrul.arslan@ed.ac.uk)*

<sup>3</sup>*EPSON Scotland Design Centre,  
The Alba campus, Livingston,  
EH54 7EG, UK*

## Abstract

*The Algebraic Codebook Search (ACS) is a computationally intensive and time consuming process used in modern speech coding techniques such as the 3G mobile network. This paper presents a new low-power generic reconfigurable platform for the Algebraic Codebook Search (ACS) function. The proposed architecture performs the ACS in about 75x times less clock cycles when compared to previous generic and application specific DSP architectures. Furthermore it gives a significant reduction in power dissipation by a factor of 9.7x compared to a DSP while there is an area overhead by a factor of 3.5x.*

## 1. Introduction

The need for better audio quality over the radio channel used in mobile technology has led the industry into the development of various audio codecs for future networks that tend to improve the quality of transmitted data over the channel, while at the same time make a better use of the available bandwidth. The most complex calculation in the Adaptive MultiRate (AMR) codec involves the Algebraic Codebook Search (ACS) that consumes approximately 36% of the total processing power of a DSP running this algorithm [1]. The ACS is constantly trying to find the optimum codeword to use for compressing the data to be transmitted over the available channel. It uses 8 modes of operation varying from 4.75 to 12.2 Kbit/s as described in the mathematical analysis given in [2] and the ANSI-C code [3].

Other implementations of the AMR codec are

described in literature but none of these presents a flexible solution while most are software based implementations. This is because such algorithms require a high flexibility that cannot be achieved in custom VLSI hardware. An FPGA implementation could provide enough flexibility and performance but the power consumption would be too high to use in portable applications.

This paper presents a novel low-power reconfigurable datapath suitable for the realization of the ACS algorithm with reduced power dissipation compared to FPGA or DSP based implementations. This platform can target various algorithms that includes convolution, FFT, DCT etc.

The rest of the paper is organized in six sections. The next section presents a brief review for previous work that has been done in this field. In section 3 an overview of the ACS theory is given, sections 4 and 5 describe the proposed architecture and present area and power results respectively. Finally, the conclusions are given in the last section.

## 2. Previous Work

A working bit exact implementation of the AMR algorithm is introduced in [4] based on a TeakLite DSP core. Since it's based on a DSP processor, the clock cycles needed to perform the calculations have an impact on power consumption. This design is not low power and has been optimized for speed only. Another approach for a fast ACELP codebook search method is presented in [5] with a 40% reduction in computational complexity. This is carried out by reducing the search trees of the algorithm. The fact though, that there is a degradation in

speech quality means that it does not comply with the 3GPP standard and thus makes it unusable by vendors.

In [6] an FPGA implementation is described by creating a DSP core. The authors use pipeline architecture and parallel execution techniques in order to increase the throughput of the core. Mostly the pipeline is used for the L\_mult function described in [3] and tries to make L\_add and L\_sub functions as fast as possible. But the fact that this design is based on an FPGA makes it unusable for mobile devices since FPGAs consume too much power by default.

The fast search method described in [7] manages to reduce the search time by a factor of 2x but decrease the speech quality by 0.5dB and at the same time the codebook search error is 4x times bigger. In [1] a low power Algebraic Codebook search accelerator is introduced using an OakDSP together with an optimized hardware but looks like it is not suitable for other kinds of algorithms since you cannot change the datapath.

### 3. Algebraic Codebook Search

The algebraic codebook is searched by minimizing the mean squared error between the input signal and the synthesized output, which is defined as,

$$\varepsilon_k = \|x - gHc_k\|^2 \quad (1)$$

, where  $\mathbf{x}$  is the target signal produced by the adaptive codebook contribution,  $\mathbf{g}$  is the codebook gain,  $\mathbf{H} = \mathbf{h}^T \mathbf{h}$  is defined as a lower triangular Toeplitz convolution matrix and  $\mathbf{c}_k$  is the algebraic code vector at index  $k$ . Alternatively we can write (1) as

$$E = \mathbf{x}^T \mathbf{x} - \frac{(\mathbf{d}^T \mathbf{c}_k)^2}{\mathbf{c}_k^T \Phi \mathbf{c}_k} \quad (2)$$

, where  $\mathbf{x}$  is the vector containing the signal  $x_2$ ,  $\mathbf{c}_k$  is the vector containing the codeword  $\mathbf{d}^T = \mathbf{H}^T \mathbf{x}_2$  is the correlation between the impulse response  $\mathbf{h}(n)$  and the target  $\Phi = \mathbf{H}^T \mathbf{H}$  is the matrix of correlations of  $\mathbf{h}(n)$ . As we can see from (2) we only need to maximize the second term  $\frac{(\mathbf{d}^T \mathbf{c}_k)^2}{\mathbf{c}_k^T \Phi \mathbf{c}_k}$  in order to minimize the error.

The equation  $\mathbf{c}_k^T \Phi \mathbf{c}_k$  can be represented as [8]:

$$E_d = \sum_{i=0}^{N_p-1} (\phi'(m_i, m_i)) + 2 \sum_{i=0}^{N_p-2} \sum_{j=i+1}^{N_p-1} (\phi'(m_i, m_j)) \quad (3)$$

, where  $m_i$  is the position for the  $i$ th pulse and  $N_p$  is the number of pulses. Equation (3) along with  $\mathbf{d}^T \mathbf{c}_k$  gives

us the mathematical representation for the hardware we need to implement.

### 4. Proposed Implementation

The overall architecture proposed in this paper is shown in Figure 1. We can distinguish the existence of a host controller, the memory and the IP core proposed for creating the required data path. The controller can be implemented using a DSP or other processor that is able to generate/calculate all previous steps prior to the ACS and store the data needed for it in the memory provided. Then the ACS IP Core generates the codeword using those data and sends it over to the host controller that continues with the speech coding process. While the codeword is calculated the host controller can do other significant computationally intensive tasks or enter a power save mode to reserve battery, ex. in case of a mobile device.

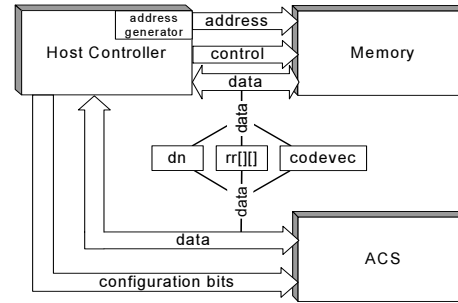
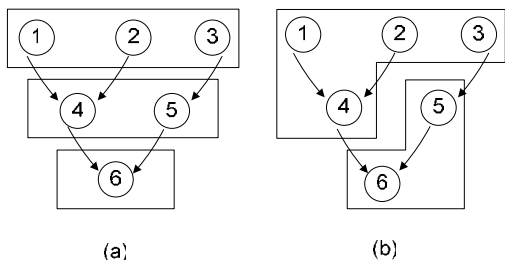


Figure 1. Overall architecture

This architecture is different from a VLIW processor because as shown in Figure 2 VLIW processors can only execute limited independent parallel instructions thus first operations 1, 2 and 3 will be executed in parallel and then operation 4 and 5 and finally operation 6, consuming 3 clock cycles. Our design can do independent and dependent instructions, under the resource constraints so you are able to execute 1, 2, 3 and 4 at the same time and do the same with 5 and 6, thus consuming only two clock cycles. This is the case in the proposed architecture described later on the paper where you can allocate the resources and execute the processes as it suits your needs.

This improvement also simplifies the design for the Finite State Machine (FSM). In the case of an operation that is repeated consecutively in the algorithm (ex.  $s = a + b$  and  $s = s + c$ ) the designer does not care on how to allocate the same resource (adder in our example) in the FSM, have counters and monitor how many clock cycles has passed, because it is treated as one operation ( $s = a + b + c$ ) that is completed in one clock cycle. The only thing that you need to be careful with is during the decision on

how the steps are going to be executed.



**Figure 2. Instruction execution between a) VLIW processor and b) our architecture**

```

i0=ipos[0]
{ for (i1=1; i1 < 5; i1=i1+1)
  { for (i2=ipos[2]; i2<40; i2=i2+5)
    { ADD
      SAC
      SAC
      SAC
      for (i3=ipos[3]; i3<40; i3 =
i3+5)
      { SAC
        SAC
        SAC
        RND
        ADD
        SAC
        SAC
        SQ
        RND
        SUB
        SUB
        CMP
      }
    }
  }
}

```

SAC: Shift Accumulate  
 RND: Round  
 CMP: Compare  
 SUB: Subtract  
 ADD: Addition  
 SQ : Square  
 MULT: Multiply

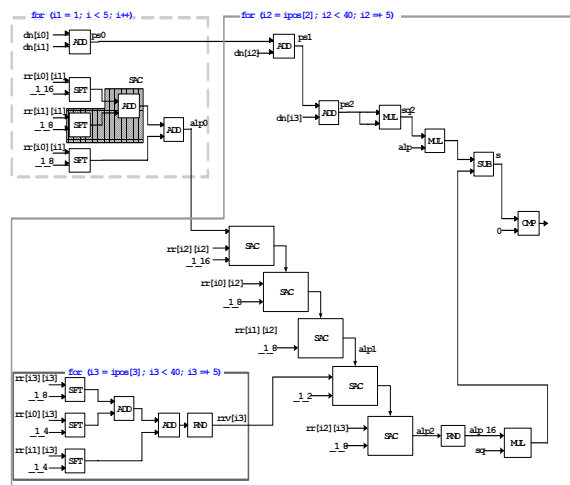
**Figure 3. Nested loop structure**

The overall loop structure for ACS is show in Figure 3 where we can see that calculations are performed in pairs of (i2, i3), (i4, i5), (i6, i7) and (i8, i9) while i0 is set into the position corresponding to the global maximum value and i1 is set to the local maximum of one track. The commands needed to execute are shown also which increase in complexity while the algorithm goes deeper to the loops. The data path, in form of a schematic for the algorithm we need to implement can be seen in Figure 4 where the various computations needed for the first two loops are shown clear with  $i = i_2$  and  $j = i_3$  in (3). We must

note that  $i_0$  is set to the local maximum by default.

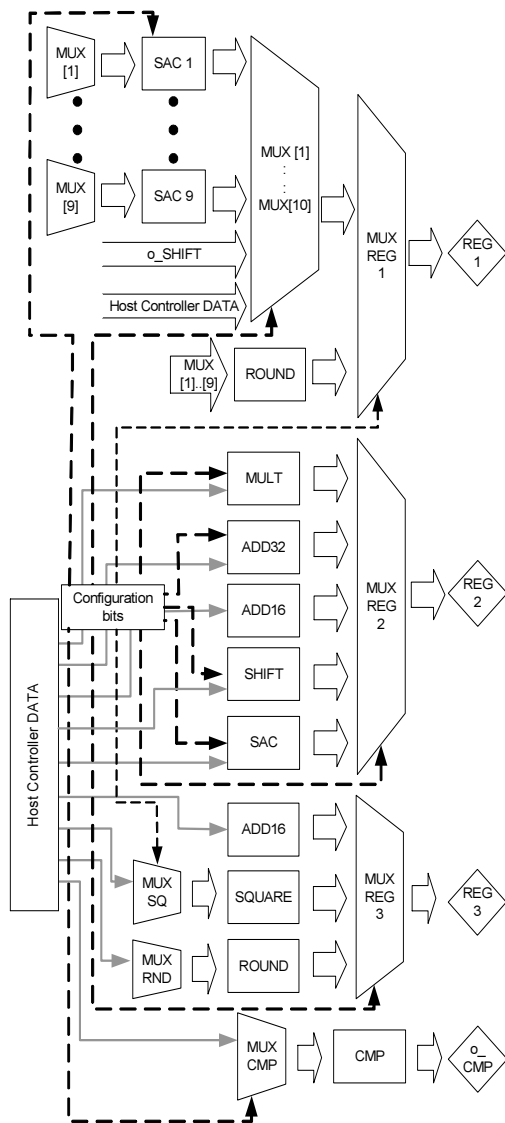
The detailed architecture proposed in this paper can be seen in Figure 5. This allows constructing a data path, which supports the operations needed to complete the calculations for (2), by just configuring the multiplexers to choose different input source for each module in each clock cycle.

In the proposed design shown in Figure 5, there are 3 general purpose registers that can be used to export data from the IP core to the host controller. The output of the comparator is used in order to decide if the value calculated is better than the previous one. The host controller can change the configuration bits of the multiplexers dynamically and thus connect various outputs with various inputs. The use of multiplexers make the frequency of the overall design low but since you implement more calculations in a clock cycle it is actually faster than previous implementations.



**Figure 4. Loop structure for i1 and (i2, i3).**

All MAC units have 16 bit inputs and a 32 bit output. Each MAC unit consists of an adder and a shifter; since all operations executed with the MAC can be converted to shift accumulate (SAC). That is easier to comprehend if we take a look in the ANSI-C code [3] and observe that the data is multiplied with fractions to scale down the numbers and thus achieve better resolution in the results. The quotation used is 16Q15 which means that 16 bit numbers are used with the 15 LSBs representing the fractional part. Because of limitations in the hardware constant multiplications would have produced too large numbers that would be difficult to handle and now that they are scaled we don't loose any resolution. Those operations will be know from now and on as SAC but if the reader wants to cross-reference with the ANSI-C code provided by [3] he will have to consider the substitution when doing so.



**Figure 5. Architecture of ACS**

MUXREG1 and MUXREG2 have ten and eight 32 bit inputs and 32 bit outputs respectively. MUXREG3 has 16 bit inputs and a 16 bit output since all modules with 16 bit outputs has been connected there. For MUXREG2 not all SAC outputs are connected since we only need half of them and most SAC operations are executed in pairs of 3, 5, 7 and 9. The shifter is used again in the place of a multiplier to scale down by a  $2^n$  factor, creating a 32 bit output from a 16 bit input.

In the architecture there is both a booth multiplier and a

square unit that is a multiplier from DesignWare™ IP library with only one 16 bit input. The booth multiplier has been used because it is fast and low power but the biggest drawback is that is quite big in size. This multiplier is used in order to perform all other multiplications that can not be converted in shift operations.

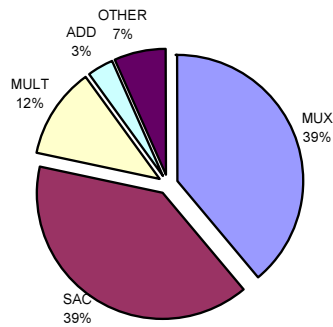
The square function could have been implemented using the same multiplier but it was decided that it would be better to use a separate unit mainly for two reasons. First because we have to wait for a clock cycle to change the configuration bits in the relevant MUX while now that it is a separate unit it can be executed on parallel and second it consumes less power than the multiplier because of the smaller size since it only has one 16 bit input and a 16 bit output compared to the multiplier that has a 32 bit output and two 16bit inputs. This reduces the switching activity by half. The rounder has a 32 bit input and a 16 bit output. It performs an addition with  $8000_H$  and a right shift until the 16 MSB bits reach the output.

The use of 9 SAC units has been selected because of the maximum number of shifting and additions needed to be performed in one clock cycle during the ACS of the AMR algorithm. We could have connected the shifters with the adders using multiplexers but that would only generate more switching activity. If the program wants to use just a shifter a separate one is provided alongside with a 16 bit and a 32 bit adder. This design also supports the use of two or more datapaths to be created simultaneously as long as the proper data are fed to the proper inputs and the outputs are droved to separate registers.

We have to point out that all units are isolated when they are not used and thus do not consume power other than static (or leakage) power. This is mainly achieved because we use multiplexers in front of the inputs for each module that change with. All modules used saturate the output constantly in  $7FFF_H$  and  $8000_H$  for upper and lower limit as noted in [3].

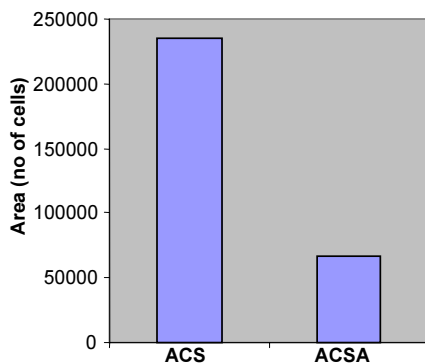
## 5. Results

Both designs were synthesized using standard Synopsys tools using the UMC0.18 technology libraries. The area coverage can be seen in Figure 6: most of the area is occupied by the SAC and MUX units (39% each respectively). This translates to 91457 and 92863 cells of a total of 235421 cells. An increase to area by a factor of 3.5x compared with the architecture described in [1] is introduced. Figure 7 provides a comparison for area between the two architectures and it must be noted that both figures do not include the area occupied by both memory and controller.



**Figure 6. Area coverage for ACS**

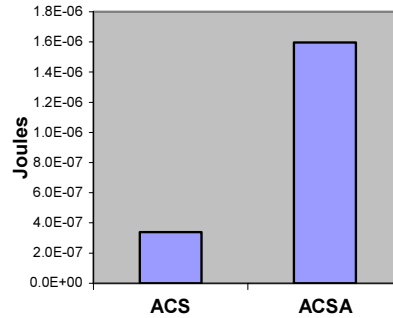
Power analysis has been performed for the datapath section only, excluding the power for memory transfers between the host controller and the IP Core. Standard ASIC tools were used to perform the calculations and the power consumption results can be seen in Figure 8 and Figure 9 in comparison with [1] and with the total power dissipation of the design respectively. From the graph in Figure 8 we can consider that most of the power is consumed in MULT, SQ, SAC2 and MUXREG2 by 22%, 17%, 10% and 8% respectively. This is expected since most of the switching activity occurs in these modules. The last two modules are constantly in operation since MUXREG2 is the multiplexer for the output that carries out most of the traffic to host controller and SAC2 is a module that is used in every loop, taking part on most operations. The first two modules perform the multiplications which are always power consuming operations.



**Figure 7. Area comparison**

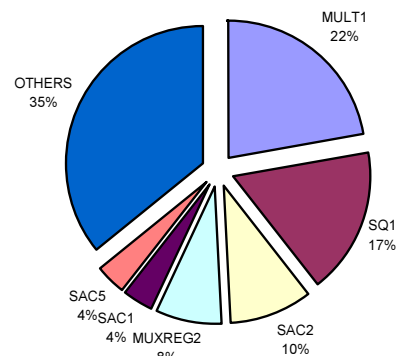
One thing that needs to be explained is how power figures for architecture in [1] have been obtained. The total number of clock cycles for 1 frame is 85744 and the design was running in 50MHz so one frame takes 1714.88E-6 sec. One frame consists of 1693 samples but

we use only 373 samples to make the comparison thus they need 376.73E-6sec and if we get an average power consumption of 4.3mW we derive to the final result.



**Figure 8. Power comparison of ACS and ACSA**

We can roughly estimate the power saved by using shifters instead of multipliers. In the case of the AMR algorithm the multiplier is used twice during a loop (MULT and SQ), while the shifter is used on average 8 times. This means that we save power that would be used for the multipliers and now only a small part of it is used for shifting without altering the final result. The total area of the design is bigger but the power is actually smaller.



**Figure 9. Power breakdown for the proposed implementation**

This happens because of three basic factors. First, the overall clock cycles needed to complete the calculations for the ACS are significantly less than the other implementations (for ex. OakDSP® requires 120.000 clock cycles [1] to complete one frame while at the same time we only require 1600). Second the active area in every clock cycle is smaller than the overall area and thus no switching activity occurs to unused hardware, finally because of the replacement of the multiplier with a shifter that consumes less power.

A comparison with other architectures can be seen in Table 1 where ACS is compared with OakDSP and ACSA.

## 6. Summary and Conclusions

This paper has presented a reconfigurable platform that can efficiently perform the Algebraic Codebook Search without degradation in speech quality in 75x less clock cycles compared to a DSP. At the same time it has enough flexibility so that the Host Controller can reconstruct the datapath and implement other types of algorithms like DCT, FFT etc. With this architecture it is possible to combine and execute up to 10 operations together in one clock cycle, saving processing time by taking advantage of the increased parallelism. For the ACS a power conservation of up to 4.7x has been achieved compared to [1] and up to 9.7x compared to OakDSP using shifters instead of multipliers where it was needed. This makes our reconfigurable core suitable for use in 3G mobile phones or other types of battery dependant devices that use algorithms to compress audio or video signals.

**Table 1. Comparison with other architectures**

	OakDSP	ACSA	ACS
Clock Cycles	120,000	87,544	1,600
Power (Joules)	-	1.6uJ	0.34uJ
Area (UMC0.18)	-	66,883.12	235,421.1

## 7. References

- [1] Kirkham, T.; Arslan, T.; Westall, F.; Crawford, D.H, "A low power datapath for algebraic codebook search targeting a generic GSM system-on-chip platform" presented at System-on-Chip, 2003. Proceedings International Symposium on, 19-21 Nov. 2003, Pages: 53 – 56.
- [2] Digital cellular telecommunications system (phase 2+) Adaptive Multi-Rate (AMR) speech coding (GSM 06.90 version 7.2.0 Release 1998) Draft ETSI EN 704 V7.2.0 (1999-12).
- [3] Digital cellular telecommunications system (phase 2+) Adaptive Multi-Rate(AMR) speech coding ANSI-C code for the AMR speech codec (GSM 06.73 version 7.3.0 Rel 1998).
- [4] K. J. Byun, H. B. Jung, M. Hahn, and K. S. Kim, "Computationally efficient implementation of AMR speech coder," presented at Image and Signal Processing and Analysis, 2003. ISPA 2003, Sept. 18-20, 2003.
- [5] K. J. Byun, H. B. Jung, M. Hahn, and K. S. Kim, "A fast ACELP codebook search method," Signal Processing, 2002 6th International Conference, vol. 1, pp. 422 - 425, 26-30 Aug. 2002.
- [6] H. Noori, H. Pedram, A. Akbar, S. Sheidaei, "FPGA

implementation of a DSP core for full rate and half rate GSM vocoders", Microelectronics, 2000. ICM 2000. Proceedings of the 12th International Conference on, Oct. 2000, pp. 273 – 276.

[7] N. K. Ha, "A fast search method of algebraic codebook by reordering search sequence" presented at Acoustics, Speech, and Signal Processing, 1999. ICASSP '99., Proceedings 1999 IEEE International Conference, 15-19 March 1999.

[8] Mandatory Speech Codec speech processing functions AMR speech codec; Transcoding functions. 3G TS 26.090 version 3.0.1.