EEEE-720: Designing a Multi-Channel ADPCM CODEC Using a Top-down Approach

Connor Goldberg, *Student Member*

***Abstract*—The use of adaptive differential pulse-code modulation (ADPCM) is a method of reducing the bandwidth of a signal, typically used for speech compression. This paper presents a design of a multi-channel ADPCM CODEC (MCAC) using a top-down approach with system on chip (SoC) and application specific integrated circuit (ASIC) design methodologies. The Amber 25 by OpenCores, an open-source 32-bit reduced instruction set computing (RISC) processor compatible with the ARM v2a instruction set, was used in conjunction with a hardware co-processor to realize the MCAC. The hardware was developed using the Cadence tool set (Design Compiler, PrimeTime, SimVision, *etc.*) and source control was manged by a Git repository. The hardware co-processor is responsible for the prediction and signal reconstruction in the ADPCM algorithm while the code running on Amber is responsible for the rest of the calculations. The ADPCM algorithm requires 8 floating point multipliers, however a single resource architecture was implemented to save area. The hardware was verified using using a combination of stimulus vectors and behavioral modeling depending on the module under test. The encoder and decoder modules of the MCAC were fully verified at RTL level; however, complete netlist verification was not completed due to time constraints.**

***Index Terms*—ADPCM, ASIC, Cadence, OpenCores, RISC, SoC.**

I. Introduction

THE use of ADPCM is a fairly efficient method of reducing the bandwidth of signals and is typically used for speech compression. ADPCM was developed at Bell Labs to improve on existing voice coding technolog [1]. The goal of the MCAC is to implement the ADPCM algorithm as per the G.726 specification written by the International Telecommunication Union (ITU) [2].

As in any digital signal processing (DSP) algorithm, an analog signal is sampled, quantized, then encoded. There are many different methods for encoding a signal. Pulse code modulation (PCM) is one such method in which analog data is represented in a digital format, usually implemented in audio applications. The rate at which a signal is sampled is dependent on the bandwidth of the signal being sampled. For speech, the frequency generally ranges from 300 to 3400 Hz [3]. To sample a signal without aliasing, the Nyquist rate of $f_s = 2W$, where W is the bandwidth of the signal, must be applied. In addition, a guard band can be included to prevent signal interference. Using the Nyquist rate along with a guard band results in a sampling frequency of $f_s = 2W + W_{GB} = (2 \times 8000kHz) + 1.2kHz = 8000kHz$. Each sample is 8-bits wide which leads to a 64 kbit/s signal (*i.e.* Digital Signal 0 or DS0).

To increase the signal-to-noise ratio (SNR) of the transmitted signal, the PCM is companded using either µ-Law or A-Law [4]. Both of these schemes are non-uniform and logarithmic in nature; they provide finer quantization for the more probable values of the speech spectrum. A-Law is primarily used in Europe while µ-Law is primarily used in North America and Japan. The MCAC must be able to account for both types of PCM.

Another method of increasing the SNR of the transmitted signal is by using either differential PCM (DPCM) or adaptive PCM (APCM). DPCM is similar to PCM however instead of quantizing the actual levels of the input signal, the difference between consecutive samples are quantized. This can result in a much smaller dynamic range assuming a slowly varying signal. In PCM and DPCM, the levels of quantization are constant; however, in APCM the quantization levels can change depending on the nature of the input signal. Combining both DPCM and APCM results in adaptive differential PCM (ADPCM) and is a much more common and efficient method of speech compression than the two former methods.

II. Theory of Operation

This section describes the ADPCM algorithm in detail and the system overview for how the algorithm is implemented.

A. Algorithm Overview

The ADPCM algorithm contains both an encoder and a decoder. Block diagrams for the encoder and the decoder are shown in Fig. 1 and Fig. 2, respectively. The input to the encoder is 64 kbit/s µ-Law or A-Law PCM, and the output is either 16, 24, 32, or 40 kbit/s ADPCM. The decoder inputs and outputs are the reverse of the encoder. The important thing to note from these figures is that there effectively is a decoder inside of the encoder. The reason for this is so the encoder can perform the same operations that the decoder will perform, and calculate differential signal (*i.e.* the error) based upon what the decoder will do. This allows the encoder to then adjust the quantization coefficients to reduce the quantization noise being introduced.

B. System Overview

The system consists of three modules in the top-level design: the encoder, decoder, and configuration module. The high-level system block diagram is shown in Fig. 3. The encoder and

C. Goldberg is with the Department of Electrical and Microelectronic Engineering, Rochester Institute of Technology, Rochester, NY 14623, USA (e-mail: connor@connorgoldberg.com).

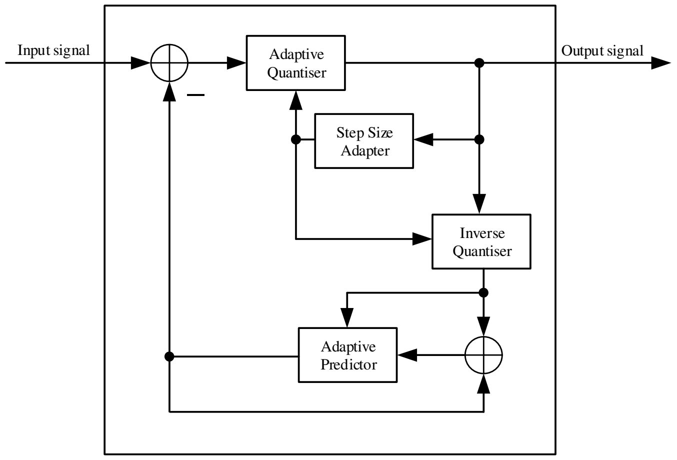


Figure 1. ADPCM encoder block diagram

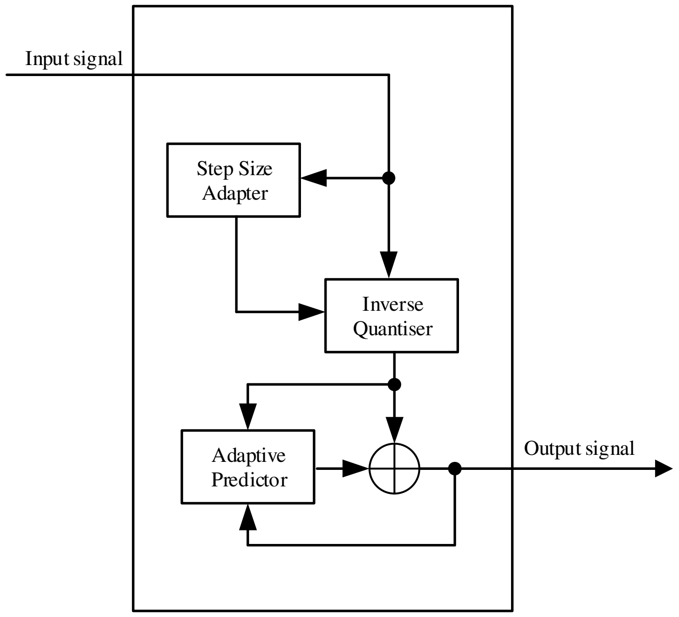


Figure 2. ADPCM decoder block diagram

decoder are responsible for the ADPCM algorithm while the decoder contains the necessary information for each of the 32 channels. Two memory cuts are located external to the system; however, everything else needed for the MCAC is internal. The MCAC is a full-duplex implementation, therefore the encoder and decoder can operate simultaneously and completely independent from each other. The PCM serial data is read in by the encoder and sent out by the decoder at a rate of 64 kbit/s. Within that data stream are 32 channels of information, 8-bits (1 byte) per channel. The rate at which the data is clocked into the MCAC is calculated by multiplying the data rate by the number of channels: $f_{serial-clk} = rate_{channel} \times N_{channels} = 64 \times 32 = 2.048$MHz. This clock is generated by the clock generation module which is instantiated in both the encoder and the decoder.

In addition to the PCM data entering the MCAC, a frame sync signal is also captured as an input. This signifies the beginning of the 32-channel frame. As soon as the frame sync

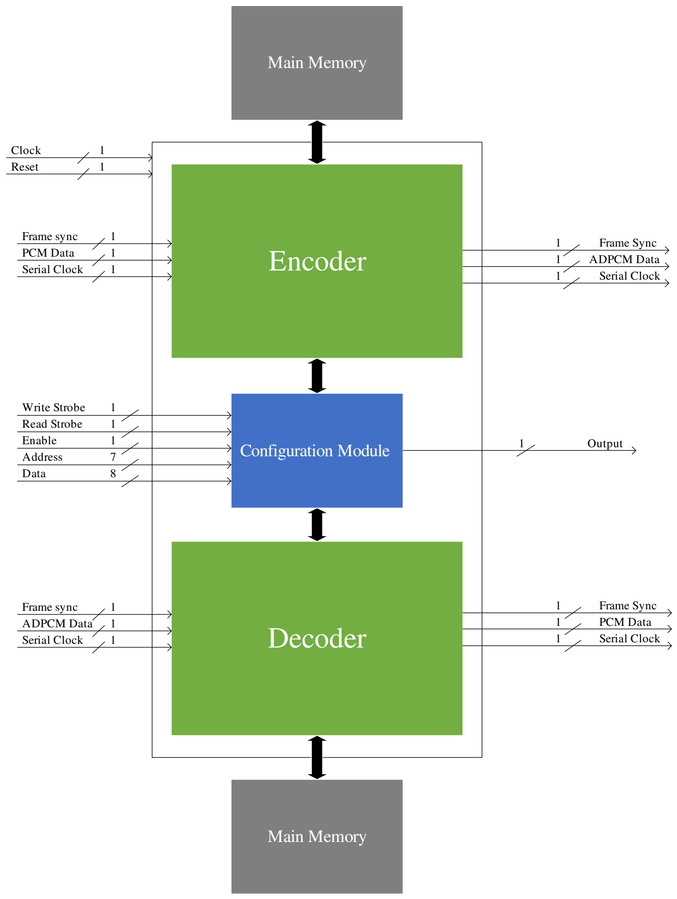


Figure 3. MCAC block diagram

signal goes high, the encoder or decoder will start operating on channel 0 again. The frame sync signal is designed to be an 8000 kHz signal to allocate 8-bits per each of the 32 channels. The time between frame syncs is found by simply calculating the period of the frame sync: $T_{frame-sync} = \frac{1}{8000kHz} = 1.25$ µs. Dividing this time by the number of channels will give the time allocated for each channel to complete the full ADPCM algorithm: $T_{channel} = \frac{1.25\mu s}{32} = 3.90625$ µs. Each channel must successfully complete the algorithm faster than this time period otherwise the data will not be ready by the next frame sync when the data needs to be written out.

III. System Architecture

A detailed block diagram of the encoder/decoder is shown in Fig. 4. This shows the mid-level modules that are instantiated in both the encoder and decoder. This section will go into more detail on each of these modules.

A. *OpenCores*

OpenCores is a community hub that contains many different open-source digital hardware projects. Several components within the Amber 25 core were used in both the encoder and decoder. The modules used are shown as the 4 left-most modules on the bottom half of Fig. 4, along with the Wishbone arbiter.

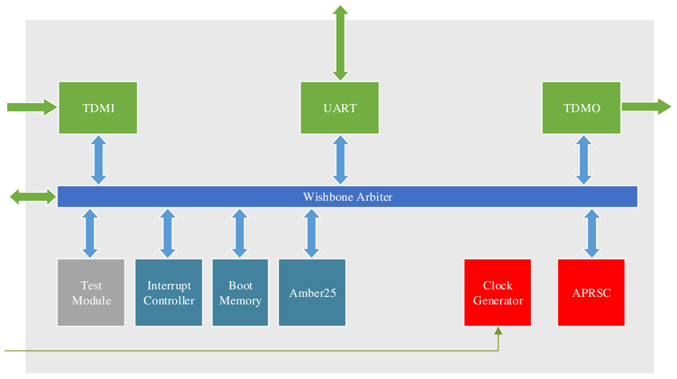


Figure 4. Encoder / decoder block diagram

1) Amber 25: The Amber 25 core is an open-source RISC processor [5]. It is based on the 32-bit ARMv2a instruction set and contains a 5-stage pipeline and also along with internal instruction and data cache units. The benefit to using this processor is that it is open-source and can be modified more easily than some other RISC processor alternatives. The entire project is written in Verilog; however, it was written to be optimized for a field-programmable gate array (FPGA). This was an issue because there was initially no reset logic in the core due to their presumption that the FPGA would handle the reset. Because of the MCAC's ASIC design methodology, reset logic had to be inserted manually into the Amber core.

2) Wishbone Arbiter: The Wishbone bus is the main method of transferring data throughout the encoder and decoder [6]. The arbiter operated based on a master-slave architecture and request-acknowledge communication scheme. The Amber core acts as the only master and it is responsible for requesting data and generating the bus cycle transactions to transfer data to/from any of the other modules. The address signal is 32-bits wide which is a multi-purposed. The upper 16-bits are used as an ID for each module while the lower 16-bits are then used inside of that module as per the its need. One interesting issue that was encountered with the Wishbone was how the data was being sent. In all of the modules interfacing to Wishbone, the data bus is 32-bits wide internal to the module; however, the data bus in Wishbone and in the Amber core is 128-bits wide. For reasons that were never completely figured out, Amber would not always read the lowest 32-bits of the data bus (*e.g.* sometimes Amber would read bits 31 to 0, sometimes it would read bits 63 to 32, *etc.*). As a quick fix, the lowest significant 32 bits of data were replicated 3 more times to fill the entire 128-bit bus so Amber would read the correct data independent of where it was looking on the bus.

3) Other OpenCores Modules: The boot memory is used to store some code that is responsible for starting the Amber core, and both the data and instruction caches. The interrupt controller is responsible for generating an interrupt to the Amber core. One issue that was encountered when attempting to set-up interrupts was that the interrupt vector table is mapped to the boot memory address space as opposed to the data memory address space. To account for this, some assembly code was written in the boot memory to jump to the correct interrupt service routine in data memory. The test module was used for a majority of the development process. Its name, however, is slightly misleading. The module does have some features to help testability (*e.g.* a random number generator), however the main reason it was being utilized was because it contained a register storing the address for the start of the program. This module was eventually removed as the program was always placed at the same address in data memory. This address was then hard-coded in the boot-loader code to remove the need of the test module.

The last module in the design from the OpenCores database is the UART module. This module is instantiated so that an external interface could potentially be designed for the encoder and decoder; however its only current use is for debugging. Any *printf* function calls made in the C code running on Amber will be sent out of the UART module. An issue was encountered when designing the test bench for the encoder and decoder relating to the UART. Along with the real UART was a UART module specifically made for the test bench to decode the UART signals for debugging. This test bench UART had an internally generated clock which was not the same as the system clock that is clocking the real UART which lead to unreadable data in the test bench. This was fixed by adding an input clock to the test bench UART instead of the internal clock generation.

B. *Clock Generator*

The clock generator module is responsible for regenerating the input frame sync to send to the output, to generate the serial clock from the input system clock, and to distribute the input system clock to the rest of the design. The serial clock was previously derived as 2.048 MHz. The system clock changed multiple times throughout the design process due to an inability to get an accurate estimation of how many cycles the algorithm would take, and therefore how fast Amber needed to run. Originally the system clock was designed to be 81.92 MHz which allowed around 320 cycles for Amber to operate per channel. At this frequency the algorithm took around 40 µs per channel which is well over the restriction of 3.9 µs. The system clock was finalized as 245.761 MHz which gives 960 Amber cycles per channel which was sufficient. The system clock is exactly 120$\times$ the serial clock. The clock for the co-processor was finalized at 40.96 MHz which is exactly 40$\times$ the serial clock. The reason the serial clock and the co-processor clock are integer divisors of the system clock frequency is so that they can be generated exactly from the system clock. One other function that this module is responsible for is the regeneration of the input frame sync signal. In addition, if the input frame sync signal is dropped, the clock generator will continue to reproduce the frame sync based off of the last input frame sync. Once another frame sync is received, the clock generator will re-synchronize the generation of this signal.

C. *TDMI and TDMO*

The time-domain multiplexed input and output (TDMI and TDMO) modules are responsible for converting serial data to parallel, and *vice versa*. TDMI will parallelize 8 sequential input bits into a single 8-bit symbol to be sent to Amber. TDMI communicates with the interrupt controller to trigger an interrupt for when the input data is parallelized and ready for read-in. TDMO contains 32 registers (one for each channel) and stores the data that is ready to be serialized and written out. TDMO constantly loops through all 32 registers and serializes the data. An interesting problem was encountered in the the channel counter for TDMO. Glitching was seen on the counter lines therefore the counter was changed to gray code. In addition to this, TDMO is also responsible for the output of the regenerated frame sync signal from clock generator.

D. APRSC

The adaptive-predictor reconstructed signal calculator (APRSC) is the co-processor to the design that is responsible for the signal estimation and reconstruction portion of the ADPCM algorithm. The major components are two filters responsible for calculating the quantization coefficients for the next sample. One of the filters is a 6^{th}order, all-zero filter which stabilizes the predicted signal and prevents oscillation from occurring. The other filter is a 2^{nd} order all-pole filter which aids the matching of the slowly varying signals. This combination of a pole and zero filter helps the signal prediction match any general input signal. The signal estimate is generally given by:

$$\hat{x}(n) = \sum_{k=1}^{P} \alpha_k(n)\tilde{x}(n-k) \tag{1}$$

where *P* is the order of the prediction and *n* is the sample. The larger the order of the prediction filter, the more samples are considered for the current signal estimation. For instance, a predictor of order one would only be able to consider one previous sample—a linear best fit line. This calculation is performed by several floating-point multiplication (FMULT) operations and then an accumulation (ACCUM) of these FMULTs. [7]

The floating-point multiplication and accumulation (FMULT/ACCUM) is one of the important pieces of the APRSC. Because a FMULT is performed 8 times (one for each order of the filter), 8 FMULT modules would normally be used. To save on registers and combinational logic, a single resource implementation of the FMULT/ACCUM was implemented. The block diagram for this design is shown in Fig. 5. The counter controls the multiplexers to select the corresponding coefficient values from the outputs of the filters. After the first 6 coefficients pertaining to the 6-zero filter are accumulated, the partial signal estimate is stored. Then once the remaining 2 coefficients from the 2-pole filter are accumulated the final signal estimate is stored.

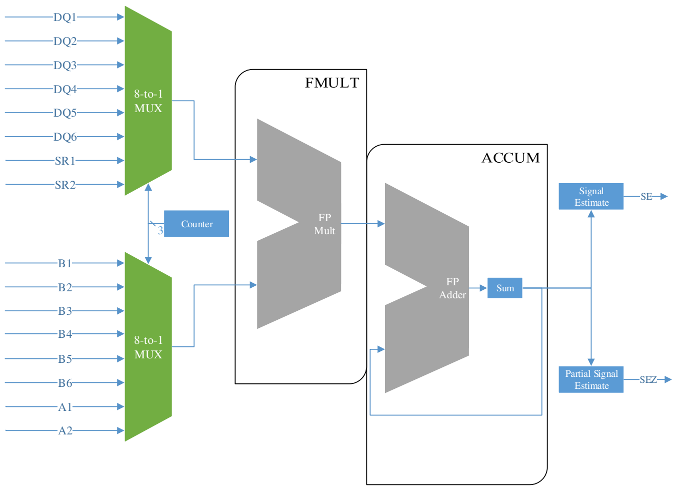


Figure 5. Floating point multiplier and accumulator

E. C Code

There were two major different C code projects that played an integral role of the design and verification of the MCAC. C programs that modeled the entire ADPCM algorithm for encoder and decoder were used to verify the system. Additionally, C programs based from the model code were written to perform the parts of the ADPCM algorithm not covered by the APRSC module.

1) C Model: A C program that modeled the ADPCM algorithm was generously provided to start the design process. This C program conformed to the American National Standards Institute (ANSI) standard for ADPCM. The requirement for this MCAC is to conform to the ITU specification therefore the C model was updated to meet the requirements set forth by the G.726 standard. This included fixing some known bugs and adding the ability to encode / decode 16, 24, and 40 kbit/s ADPCM. The ITU provided their own stimulus vectors to fully verify the G.726 implementation of ADPCM [8]. This description was used to fully verify the C model. The C program was written to model the specification therefore it contained functions and variables for all of the logic blocks within the ADPCM algorithm. These were then used to generate stimulus vectors to verify the Verilog code of the APRSC. The next step in preparing the C model was adding the ability for 32 channels. This was then used to generate top-level test vectors for MCAC.

2) C Firmware: The C code was modified to be cross-compiled for the Amber processor. Because the Amber 25 supports the ARM v2a architecture the GNU compiler can be used to target the architecture. All C code pertaining to OS specific tasks was removed along with the calculations that the APRSC is responsible for. Additionally, code was written to be able to read and write to the peripherals (TDMI, TDMO, APRSC, *etc.*). After reaching the point where the execution of the C Code within encoder/decoder was possible, the first real timing analysis was performed. The timing analysis showed that the algorithm required around 40 µs to complete for each channel. To decrease the execution time by around $10\times$, a substantial amount of code optimization was performed in addition to increasing the frequency of the system clock.

The first optimization implemented was flatten the code by removing the function hierarchy. This greatly reduced the number of jumps performed which gave a large speedup due to less stall cycles by the processor and a longer execution period in which the pipeline is filled. After performing this optimiza-

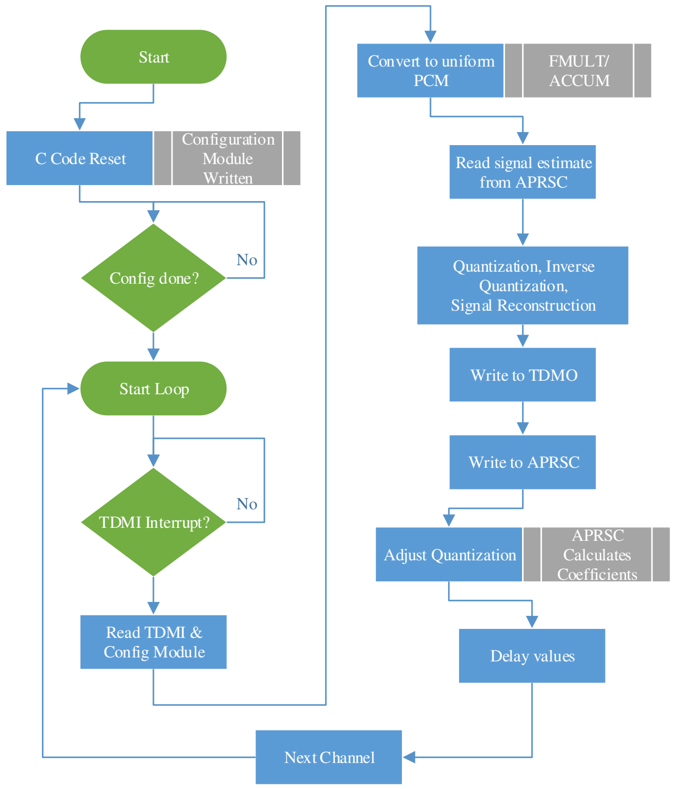

Start
C Code Reset
Configuration Module Written
Config done?
No
Start Loop
TDMI Interrupt?
No
Read TDMI & Config Module
Next Channel
Convert to uniform PCM
FMULT/ ACCUM
Read signal estimate from APRSC
Quantization, Inverse Quantization, Signal Reconstruction
Write to TDMO
Write to APRSC
Adjust Quantization
APRSC Calculates Coefficients
Delay values

Figure 6. Encoder algorithm flow

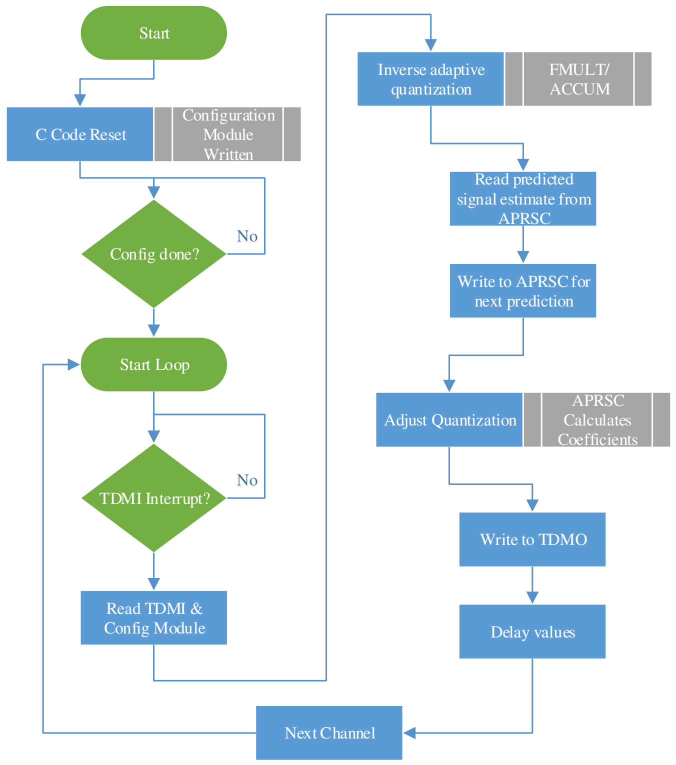

Start
C Code Reset
Configuration Module Written
Config done?
No
Start Loop
TDMI Interrupt?
No
Read TDMI & Config Module
Next Channel
Inverse adaptive quantization
FMULT/ ACCUM
Read predicted signal estimate from APRSC
Write to APRSC for next prediction
Adjust Quantization
APRSC Calculates Coefficients
Write to TDMO
Delay values

Figure 7. Decoder algorithm flow

tion the execution time was roughly 15 µs per channel. To further decrease this to be able to meet the 3.9 µs requirement, several functions were converted to look-up tables (LUTs). Due to the large memories that were being used, there was ample room to create large LUTs for computation intensive calculations. From the original encoder C code to the fully optimized code, the number of addresses used for the program increased from 11,880 to 537,180. This optimization, although increasing code size, successfully lowered the execution time enough to meet the timing requirements.

One issue encountered with the fully optimized code was that for a certain range of LUT values, the value retrieved from the table would not actually be the correct value. The underlying reason for this failure was not completely determined; however, the issue was fixed through bypassing the LUT for that specific range of values. One possibility for the failure is that the cache was failing to return the correct value due to the immense size of the LUTs.

F. Algorithm Flow

The algorithm flow charts for the encoder and decoder are shown in Fig. 6 and Fig. 7, respectively.

The green blocks in the charts represent flow control, the blue objects represent operations as performed by the C Code running on the Amber core, and the gray objects represent parallel operations in hardware.

IV. Implementation

This section will expand upon the various methodologies used in design, verification, and synthesis.

A. Design Methodology

The top-down methodology was used to design the MCAC. The design started with the high-level system descriptions as described by the G.726 specification and was split into many separate problems to approach. The system was then implemented from the bottom-up by creating the lowest-level modules first, then working up the hierarchy. Additionally, ASIC and SoC methodologies were used in the design.

B. Verification Methodology

Both stimulus vectors and behavioral modeling was used to verify the various hardware modules. For hardware modules also represented in the C model, the stimulus vector methodology was used for the verification. Known inputs were applied and the output data generated by the unit under test (UUT) was checked against the corresponding known output vector. For modules such as TDMI and TDMO that have no representation in the C model, behavioral modeling was used to verify the UUT. This consisted of applying a random input to the UUT and using test bench logic to model the functionality of the UUT and then testing against the real output of the module. For the the APRSC, encoder, decoder, and MCAC verification, a database was used to track the verification status. The run-time for a single test vector pass ranged from around 30 minutes to over 24 hours depending on which of the 128 tests were used from the appendix II of G.726 [8]. There would be no feasible way of tracking the status of all 128 tests, especially in a team-oriented effort, without the use of a database-type tracking system.

C. Synthesis Methodology

The design process initially began with the use of a 180nm library. When the system clock was increased to 245.761 MHz, there were timing violations in the design. The library was switched to a 32 nm library; however, this library was not of very high quality. The finalized library used in synthesis is a 65 nm library from Taiwan Semiconductor Manufacturing Company (TMSC). Two different synthesis options are used: RTL logic synthesis, and full-scan design for testability (DFT) insertion. DFT insertion synthesizes scan chains throughout the design along with multiplexing each module's input clock with the system clock. The DFT insertion reports on the test coverage percentage of the design. This signifies how much of the final design is guaranteed to be testable therefore a higher percentage is desired.

V. Results

This section reports the results from the various output log files generated by the synthesis tools and the PrimeTime timing analysis tools.

A. MCAC Results

The MCAC results are shown in Table I. The total cell area without memories in this table is the sum of all of the memories subtracted from the total cell area. The maximum frequency is calculated from the worst case slack and represents the maximum frequency that can still give the desired performance. Table II and Table III show the hierarchical results of the pre and post-scan netlist synthesis, respectively. The dynamic power expressed in these tables is the result of summing the switching power with the internal power. This information was gathered from the logs resulting from the Design Compiler synthesis. Table IV shows the worst case timing path for both the pre and post-scan netlist. The slack is time between when the data needs to be ready for the clock edge, and when the data actually arrives. Positive slack is good because it means the data is ready to be clocked into the register when the clock edge arrives. For both pre and post-scan the timing paths were met.

Table I
MCAC RESULTS

Metric	Pre-scan netlist	Post-scan netlist
Total cell area $(\mu m)^2$	3123285.242	3188138.154
Boot memory area* $(\mu m)^2$	412581.7125	412584.5925
Data cache memory area* $(\mu m)^2$	508024.3238	509061.4837
Instruction cache memory area* $(\mu m)^2$	506440.6837	510401.7637
Total cell area without memories $(\mu m)^2$	269191.8025	324042.4741
NAND2 area $(\mu m)^2$	1.44	1.44
Number of Gates	186938.7517	225029.4959
Total Number of Cells	2009	6246
Test coverage**	N/A	93.49%
Timing analysis coverage**	N/A	48%
Worst case slack**	N/A	-1.09E-09
Maximum frequency**	N/A	9.21E+08

* The area is doubled due to this module being instantiated in both the encoder and decoder
** These results were gathered from the PrimeTime timing analysis reports of the DFT insertion

B. Limitations and Extensions

One of the main limitations of this design is the communication between modules in the encoder and decoder. Due to the master-slave architecture of the Wishbone arbiter, the Amber core must initialize the transaction for data to move. One addition that could improve efficiency in this area would be a direct memory access (DMA) controller or similar. This could bypass the arbiter for transactions to reduce the number of instructions performed by the Amber core. If implemented this could potentially allow for a slower system clock frequency.

VI. Discussion

This section will contain the current state of the design along with system limitations and ways for future extensions onto the system.

A. Current State

The RTL code for the MCAC has been fully completed. The encoder and decoder both have completely passed all 128 vector stimuli for RTL (logical) simulation. A partial netlist verification has been performed successfully for encoder, decoder, and the MCAC; however, due to time-restraints a complete netlist verification has not been completed.

VII. Conclusion

Designing a system using a top-down design approach can be very effective if done accurately. Proper timing estimations were not completed early on in the project which led to poor clock frequency choices. Making design choices early in the design process is difficult yet necessary for a successful design. Due to the poor timing estimations early in this design, clock frequencies changed throughout the design process causing various troubles along the way. Had issues such as these been alleviated earlier on in the design process, there may have been enough time to fully verify the MCAC, encoder, and decoder at netlist level. The MCAC, encoder, and decoder fully passed all RTL simulations and passed several netlist simulations providing a partial verification. Many issues were encountered throughout the process and different solutions were applied to each one. Given enough time, the MCAC would be fully verified at RTL and netlist level to prove its functionality.

Acknowledgment

This project was completed with the invaluable guidance, support, and assistance from Mark A. Indovina.

Appendix

There were many aspects of this project that went well, and many that did not go so well. This project was by far the

Table II
PRE-SCAN NETLIST: HIERARCHICAL RESULTS

Module	Area $(\mu m)^2$	Percent of total area	Dynamic power (mW)	Leakage power (nW)	Total power (mW)
MCAC	3123285.242	100	27.513	9280	27.522
Decoder	1559874.121	49.9	13.6	4600	13.605
Encoder	1559360.761	49.9	13.47	4570	13.475
Amber*	1066932.488	68.4	1.916	1950	1.918
Boot Memory*	412581.7125	26.4	0.06805	0.828	0.0681
APRSC*	60842.8806	3.8	8.514	2100	8.517
UART*	5857.2001	0.4	1.0071	165.888	1.008
TDMO*	4632.8401	0.2	0.7748	131.707	0.775
Configuration Module	4049.2801	0.1	0.44	112.927	0.44
Interrupt Controller*	2332.8	0.2	0.3919	63.094	0.392
Wishbone*	893.52	0	0.0252	20.981	0.0252
TDMI*	826.2	0	0.14207	24.431	0.142
Clock Generator*	367.56	0	0.07356	12.456	0.0736

* The percent of total area is doubled due to this module being instantiated in both the encoder and decoder

Table III
POST-SCAN NETLIST: HIERARCHICAL RESULTS

Module	Area $(\mu m)^2$	Percent of total area	Dynamic power (mW)	Leakage power (nW)	Total power (mW)
MCAC	3188138.154	100	2.08947	8830	2.099
Decoder	1592987.277	50	1.035	4390	0.905
Encoder	1590379.797	49.9	1.004	4330	0.736
Amber*	1079934.248	67.8	0.64	1920	8.518
Boot Memory*	412584.5925	25.8	0.00149	0.68	N/A
APRSC*	77785.9172	4.8	8.87	1930	8.872
UART*	6878.5199	0.4	1.1037	162.578	1.103
TDMO*	5456.5199	0.4	0.9384	133.172	0.939
Configuration Module	4769.9999	0.1	0.4576	111.11	0.458
Interrupt Controller*	2738.8799	0.2	0.5245	64.707	0.525
TDMI*	997.2	0	0.15394	22.785	0.154
Wishbone*	898.56	0	0.02724	21.2	0.0273
Clock Generator*	432.36	0	0.07806	12.007	0.0781

* The percent of total area is doubled due to this module being instantiated in both the encoder and decoder

Table IV
WORST CASE PATH RESULTS

	Pre-scan netlist	Post-scan netlist
Path startpoint	ENC_INST0/amber_inst0/u_decode/o_load_rd_reg_3_	ENC_INST0/amber_inst0/u_write_back/mem_read_data_valid_r_reg
Path endpoint	ENC_INST0/amber_inst0/u_execute/u_register_bank/r12_reg_30_	ENC_INST0/amber_inst0/u_execute/o_daddress_reg_31_
Slack (ns)	0.0	0.00120

best and most involved learning experience that the author has experiences thus far at RIT. The author learned that working on a team is not always a pleasurable or productive experience; however, compromises must be made to meet deadline requirements. The author learned much about the life-cycle of a complete design, how to more accurately represent hardware using Verilog, and the importance of code-reviews.

REFERENCES

[1] J. R. BODDIE, J. D. JOHNSTON, C. A. MCGONEGAL, J. W. UPTON, D. A. BERKLEY, R. E. CROCHIERE, AND J. L. FLANAGAN, "DIGITAL SIGNAL PROCESSOR: ADAPTIVE DIFFERENTIAL PULSE-CODE-MODULATION CODING," *The Bell System Technical Journal*, VOL. 60, NO. 7, PP. 1547–1561, SEPT 1981.

[2] *40, 32, 24, 16 kbit/s ADAPTIVE DIFFERENTIAL PULSE CODE MODULATION (ADPCM)*, ITU RECOMMENDATION G.726 STD., 04 1991.

[3] M. I.-H. J. E. I. E. SUN, L., *Guide to voice and video over IP*, 1ST ED. SPRINGER-VERLAG LONDON, 2013.

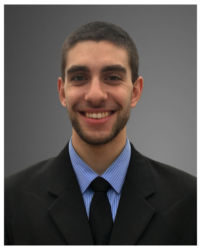
[4] *PULSE CODE MODULATION (PCM) OF VOICE FREQUENCIES*, ITU RECOMMENDATION G.711 STD., 04 1991.

[5] *Amber 2 Core Specification*, STD., 2015.

[6] *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, STD., REV. B4, 2010.

[7] M. A. JAY REIMER, MIKE MCMAHAN, "32-KBIT/S ADPCM WITH THE TMS32010," TEXAS INSTRUMENTS, DIGITAL SIGNAL PROCESSING SOLUTIONS, TECH. REP., 1989.

[8] *Description of the digital test sequences for the verification of the G.726 40, 32, 24 and 16 kbit/s ADPCM algorithm*, ITU RECOMMENDATION G.726 APPENDIX II TEST VECTORS STD., 03 1991.



Connor Goldberg is currently enrolled in the BS/MS dual degree program at Rochester Institute of Technology, New York. He is seeking a masters of science in electrical engineering with a focus in digital design, and embedded software design.