

## AN ABSTRACT OF THE THESIS OF

Richard S. Lewison for the degree of Master of Science in Electrical and Computer Engineering presented on July 14, 1997. Title: A Delta-Sigma Pulse Width Modulator with Pulse Dithering.

Redacted for privacy

Abstract approved: .

David J. Allstot

Pulse Width Modulation (PWM) has been used extensively for motor control, DC-AC converters, DC-DC converters and in audio applications. The conventional method of generating a pulse width modulated signal involves generating an accurate sawtooth or triangle wave using analog circuits. In CMOS, being analog circuit intensive puts extra constraints on the fabrication process used for manufacture, thus requiring a higher cost than digital CMOS. Delta-Sigma Modulation on the other hand is analog intensive, but requires only quality capacitor matching, which can be obtained with current digital CMOS processes. This thesis describes a method to generate a digital PWM with pulse dithering using a Delta-Sigma modulator. The Delta-Sigma modulator provides accuracy and allows the circuit to be implemented in a digital CMOS process, while the pulse dithering in the PWM spreads out harmonic noise generated from the PWM fundamental frequency.

©Copyright by Richard S. Lewison  
July 14, 1997  
All Rights Reserved

A Delta-Sigma Pulse Width Modulator with Pulse Dithering

by

Richard S. Lewison

A THESIS

submitted to

Oregon State University

in partial fulfillment of  
the requirements for  
the degree of

Master of Science

Completed July 14, 1997  
Commencement June 1998

Master of Science thesis of Richard S. Lewison presented on July 14, 1997

APPROVED:

Redacted for privacy

\_\_\_\_\_  
Major Professor, representing Electrical and Computer Engineering

Redacted for privacy

\_\_\_\_\_  
Chair of Department of ~~Electrical~~ and Computer Engineering

Redacted for privacy

\_\_\_\_\_  
Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for privacy

\_\_\_\_\_  
Richard S. Lewison, Author

## ACKNOWLEDGMENT

The author would like to acknowledge the support and encouragement of Professor David Allstot of Oregon State University throughout the development of the DSPWM. The author would also like to acknowledge Professor Wojtek Kolodziej of Oregon State University for introducing me to Delta-Sigma Modulation and for his initial motivation to explore remote uses of Delta-Sigma Modulation. Finally, the author would like to express many thanks to Dr. Dmitry Kosterev of the Bonneville Power Administration for his brainstorming during the early development of the DSPWM.

## TABLE OF CONTENTS

	<u>Page</u>
1 INTRODUCTION .....	1
2 BACKGROUND .....	2
2.1 Conventional PWM .....	2
2.1.1 Modulation Description .....	2
2.1.2 Applications .....	3
2.1.3 Noise Characteristics .....	4
2.2 Delta-Sigma Modulation (DSM) .....	7
2.2.1 Modulation Description .....	7
2.2.2 Noise Characteristics .....	7
2.2.3 Mathematical Model of DSM .....	8
2.2.4 Applications .....	10
3 DELTA-SIGMA PULSE WIDTH MODULATION (DSPWM) .....	13
3.1 DSPWM Block Diagram .....	14
3.1.1 DSM .....	14
3.1.2 Accumulator .....	14
3.1.3 Pulse Inverter .....	16
3.1.4 PWM Formatter .....	16
3.1.5 Random Offset Generator .....	18
3.2 Pulse Dithering and Inverting .....	19
4 PROOF OF CONCEPT .....	21
4.1 Model .....	21
4.2 Simulation Results .....	21
5 CIRCUIT IMPLEMENTATION .....	23
5.1 Model .....	25
5.1.1 Analog Model .....	26
5.1.1.1 DSM Top-Level Block .....	26
5.1.1.2 OPAMP .....	27
5.1.1.3 Comparator .....	30

## TABLE OF CONTENTS (Continued)

	<u>Page</u>
5.1.1.4 CMOS Switch .....	30
5.1.1.5 Current Source .....	31
5.1.2 Digital Model .....	32
5.2 Simulation Results .....	32
6 CONCLUSION .....	35
REFERENCES .....	36
APPENDICES .....	37
Appendix A Matlab Source Code Files .....	38
Appendix B Verilog HDL Source Code Files .....	48

## LIST OF FIGURES

	<u>Page</u>
Figure 1: Diagram of Pulse-Width Modulation .....	2
Figure 2: Example of Pulse-Width Modulation (PWM) waveform .....	3
Figure 3: A Switch-Mode Step-Down (Buck) Converter .....	4
Figure 4: Frequency Spectrum of PWM .....	5
Figure 5: Mechanism for Harmonic Noise Propagation .....	6
Figure 6: Block Diagram of a Delta-Sigma Modulator .....	8
Figure 7: DSM Typical Waveform .....	9
Figure 8: Switched-Capacitor Delta-Sigma Modulator (first order).....	10
Figure 9: Noise Frequency Spectrum of Simulated DSM .....	11
Figure 10: Z-Domain Equivalent Block Diagram of the DSM.....	12
Figure 11: DSPWM Block Diagram.....	13
Figure 12: DSPWM Detailed Block Diagram .....	14
Figure 13: Accumulator Schematic Diagram .....	15
Figure 14: Schematic Diagram of the Pulse Inverter.....	17
Figure 15: Schematic Diagram of the Random Offset Generator.....	18
Figure 16: Dithered & Inverted Waveform Example .....	19
Figure 17: Matlab Simulation Results .....	22
Figure 18: DSPWM System Inputs, Outputs, and Interconnects.....	24
Figure 19: Schematic of DSM .....	27
Figure 20: Schematic Diagram of Opamp .....	28
Figure 21: CMOS Switch Schematic Diagram.....	30
Figure 22: Current Source Schematic Diagram.....	31
Figure 23: Schematic Diagram of Digital PWM Block.....	33
Figure 24: Spectral Plots of Simulation Using 4kHz Input signal.....	34



## LIST OF TABLES

	<u>Page</u>
Table 1: PWM Period and Frequency vs. SW[3:0] Using a 20MHz System Clock...	25
Table 2: Optimization Objectives for Opamp Design.....	29
Table 3: Opamp Device Sizes (in $\mu\text{m}$ ) .....	29

# A Delta-Sigma Pulse Width Modulator with Pulse Dithering

1

## INTRODUCTION

Industry has long used Pulse-Width Modulation (PWM) to drive motors, to facilitate DC-DC/AC conversion, and in many other similar applications [1]. Because of non-zero switching time in power switching devices, the number of digital transitions per unit time typically determines the thermal power dissipation of systems using these devices. PWM is ideally suited for these types of applications because it has a fixed number of digital transitions per unit time. Along with the advantage of the fixed frequency of PWM, is the problem of harmonic noise from the PWM fundamental switching frequency [1]. Overcoming the problem of harmonic noise has been addressed in recent literature [3], [4], [5]. Most methods for spreading harmonic noise involve dithering the PWM fundamental period, or introducing extra transitions in the PWM waveform to tailor the frequency spectrum by removing specific harmonics. The latter method does not retain the PWM feature of a fixed number of digital transitions per unit time. This paper introduces a PWM design that spreads harmonic noise in the frequency domain by dithering the PWM pulses within a fixed period, while maintaining a fixed number of digital transitions per unit time. This is facilitated by combining Delta-Sigma Modulation (DSM) with a digital PWM block to create a discrete-time Delta-Sigma Pulse Width Modulator (DSPWM).

## 2

## BACKGROUND

This section covers some background issues leading up to the development of the DSPWM including the fundamental concepts, applications, and noise characteristics of conventional PWM and DSM.

### 2.1 Conventional PWM

This section describes the fundamentals of conventional PWM. The noise characteristics and typical applications of PWM are also explored.

#### 2.1.1 Modulation Description

Conventional PWM uses a triangle or sawtooth reference waveform to compare against an incoming analog signal. The digital result of this comparison is the PWM output. Figure 1 below shows a block diagram of pulse-width modulation with an input signal and a sawtooth reference waveform driving a comparator, where the output of the comparator is the PWM output. PWM is simple, closely approximates the input signal (within the frequency band of interest), and has a fixed number of digital transitions per unit time.

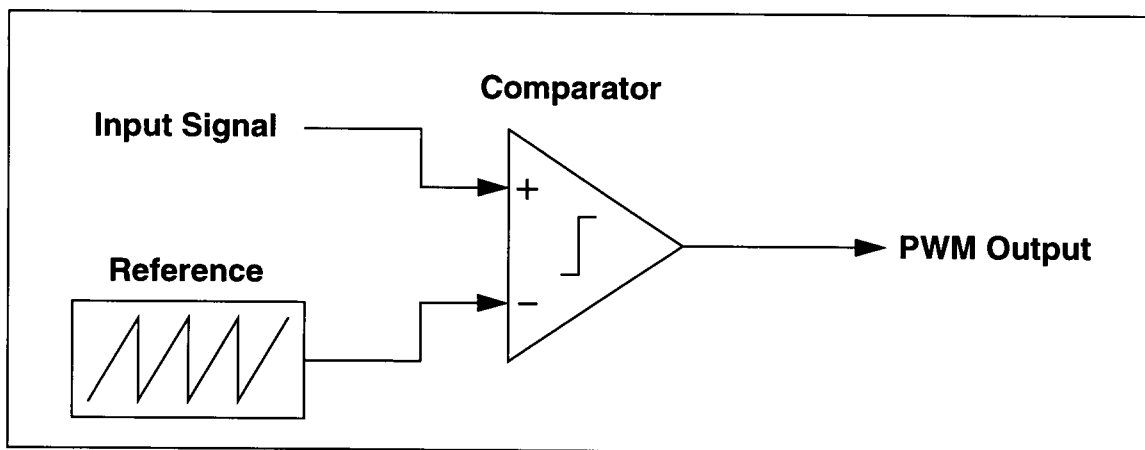


Figure 1: Diagram of Pulse-Width Modulation

An example of a PWM waveform is illustrated in figure 2 below. Creating a precise analog triangle/sawtooth waveform is typically done using an RC charge-discharge circuit employing precision resistors, capacitors and current sources. This is not a problem when using discrete resistors and capacitors, but for integrated circuits, on-chip resistors and capacitors are hard to control and can exhibit non-linear behavior when fabricated using digital-only IC processes. A delta-sigma modulator can be used to create a similar PWM waveform, without requiring a triangle/sawtooth waveform. This will be discussed later in chapter 3.

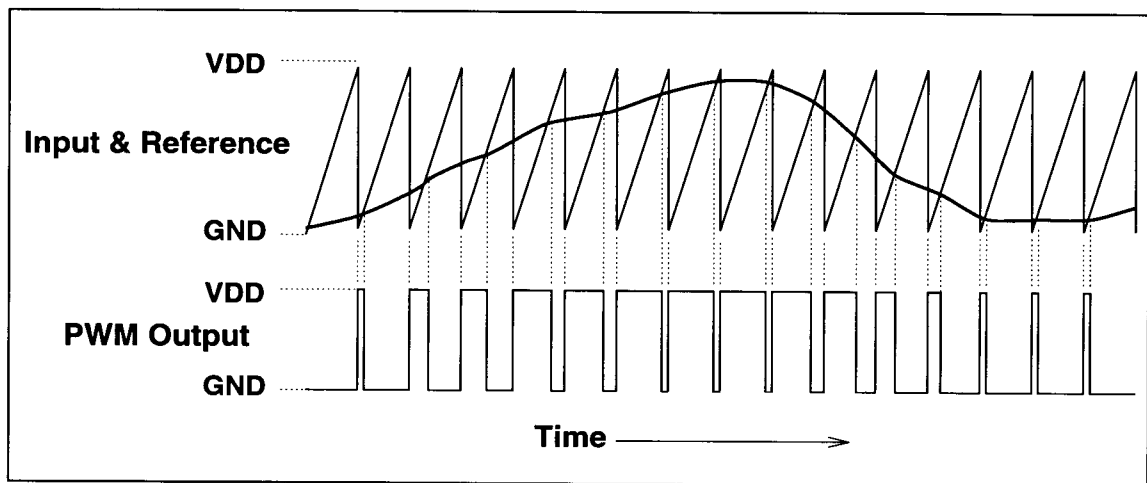


Figure 2: Example of Pulse-Width Modulation (PWM) waveform

### 2.1.2 Applications

PWM is used extensively in power electronics applications. Typical applications include power inverters (DC-AC converters) and DC-DC switch-mode converters (buck, boost,...). These converters operate by alternately opening and closing a switch between a DC voltage reference and complex impedance network to generate a desired output waveform using PWM as the switching control signal. A DC step-down (buck) converter is shown in figure 3 below. The switches used are typically Insulated Gate Bipolar Tran-

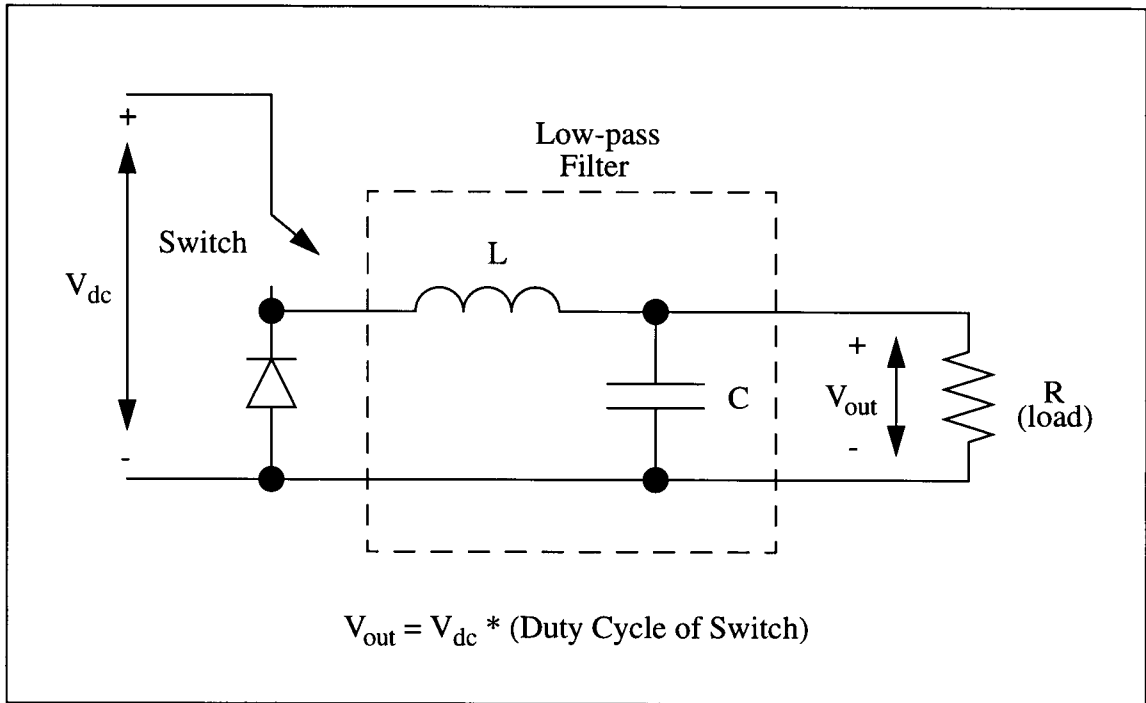


Figure 3: A Switch-Mode Step-Down (Buck) Converter

sistors (IGBTs), thyristors, or power MOSFETs. Ideally an ON state consists of a zero voltage drop across the switch, and an OFF state consists of a zero current through the switch. Either of these two cases results in a zero-power condition. When switching, however, there is a finite period of time when the voltage across the switch and the current through the switch are both non-zero. Each switching event therefore results in a finite amount of energy that needs to be conducted or radiated away from the switch.

### 2.1.3 Noise Characteristics

The fixed-frequency, varying duty-cycle rectangular waveform of conventional PWM results in a significant amount of harmonic noise [1], [3], [4], [5], [6]. This is illustrated in Fig. 4 below with simulation results of a conventional PWM, where the fundamental switching frequency is 10kHz and the input signal is a 1kHz sine wave. For this simulation a sampling rate of 1MHz was used and 100,000 samples were used to calculate the

frequency spectrum for each graph. Figure 4 shows that there is a significant amount of harmonic noise at frequencies not too far above the input signal frequency. This out of band energy from harmonic noise can result in significant electromagnetic interference (EMI) if the receiving system cannot reject signals above the frequency band of interest.

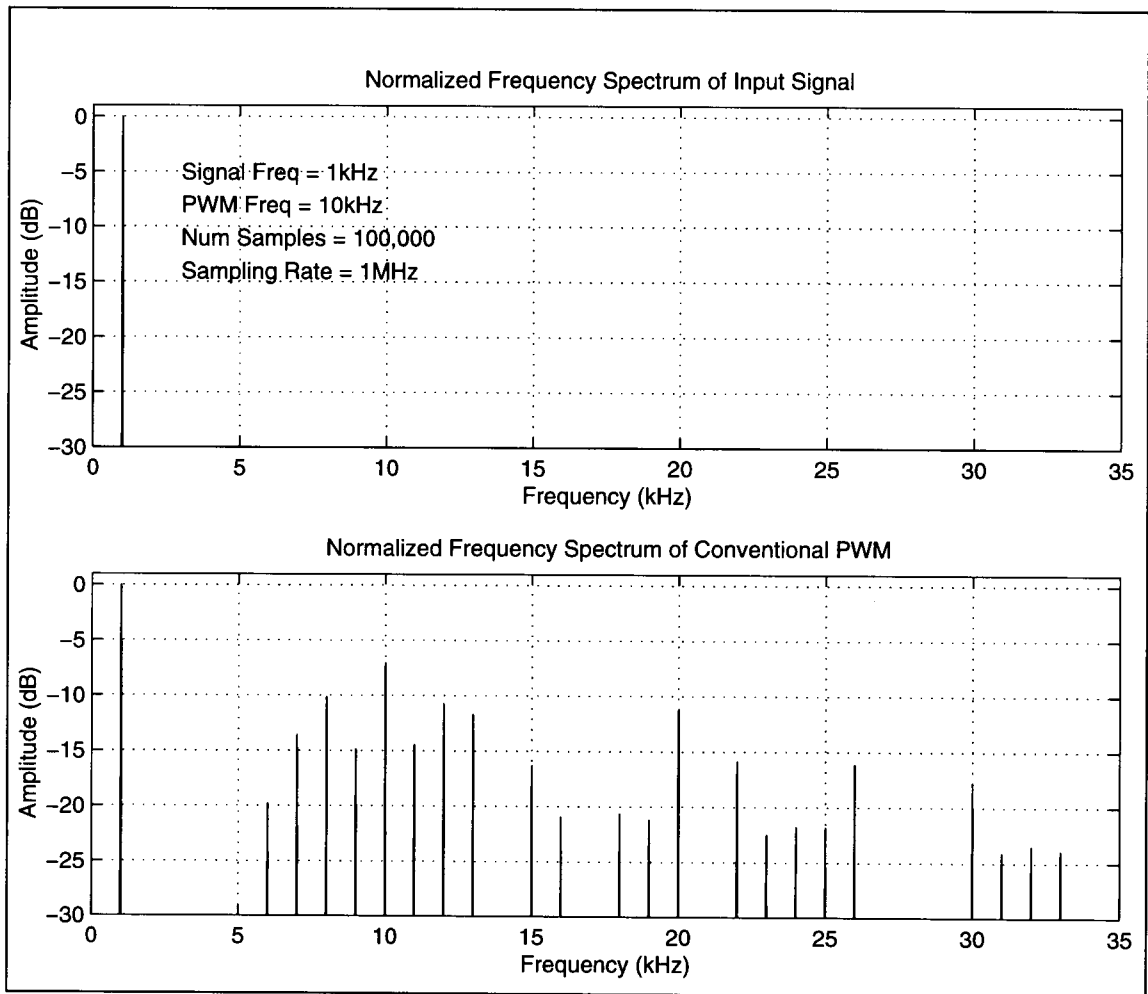


Figure 4: Frequency Spectrum of PWM

The mechanism of harmonic noise interference is shown in figure 5 below. In this figure noise can be radiated into the environment if the PWM signal is a high-current signal as is the case for driving motors. This radiated noise can interfere with other circuits that

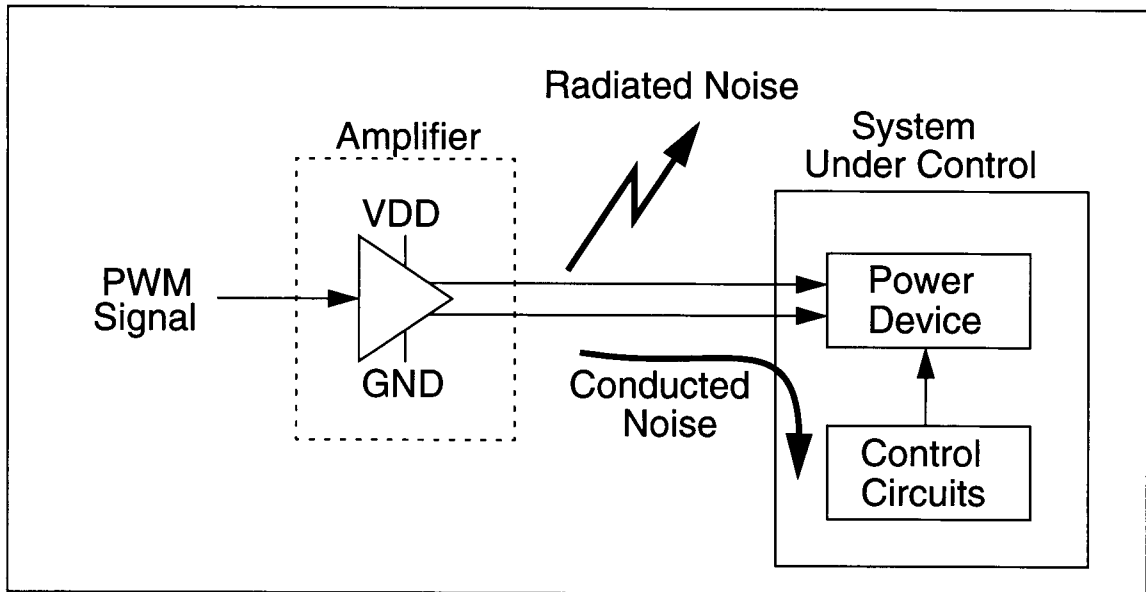


Figure 5: Mechanism for Harmonic Noise Propagation

may be sensitive to the frequencies of the radiated noise. This could be a wireless receiver or an unshielded microprocessor circuit board. Another way that harmonic noise can be a problem is through conduction. Harmonic noise can couple to system power supplies through stray capacitance and inductance. This can contaminate the power supply, resulting in power supply noise interfering with sensitive control electronics elsewhere in the system. This situation could arise in power inverters or DC/DC switch-mode converters where there is a significant amount of power device switching activity in close proximity to sensitive control electronics. Conventional methods for combatting these two mechanisms of noise propagation are shielding for radiated noise and filtering for conducted noise. Both of these methods involve added manufacturing costs to the system.

## 2.2 Delta-Sigma Modulation (DSM)

The fundamentals of Delta-Sigma Modulation (DSM) are reviewed in this section. This review consists of a brief description of DSM followed by a discussion of some DSM applications, and finally some discussion of the noise characteristics of DSM.

### 2.2.1 Modulation Description

DSM is an oversampling method for converting an analog signal to a digital waveform. Unlike conventional Nyquist sampling systems, DSM must be sampled at a rate much higher than Nyquist (twice the highest frequency component of the signal). The rate of sampling relative to the Nyquist rate is referred to as the oversampling ratio and is herein denoted by  $R$ . In practical systems, the value of  $R$  is typically greater than 100. The system-level block diagram for DSM is shown in figure 6 below. A typical DSM waveform is shown in figure 7 for a 1kHz sine wave input signal modulated at 1MHz. Notice that DSM is inherently a discrete-time system with a digital output, but not necessarily a digital system since the input is (in our case) an analog signal. In this thesis we will be referring to DSM as a discrete-time system implemented in switched-capacitor technology. A switched-capacitor implementation of a DSM is illustrated in figure 8. Note that since this is a switched-capacitor implementation, good capacitor matching is required for good DSM performance.

### 2.2.2 Noise Characteristics

DSM has the unique characteristic of shaping the quantization noise spectrum. Where a conventional sampling system may have a constant (quantization) noise floor due to a uniform distribution of quantization noise of between  $-1/2$  and  $+1/2$  lsb for all frequencies, DSM has a shaped noise floor that is minimum at low frequencies and increases with frequency. This is illustrated in figure 9. Note that at low frequencies, the noise floor of DSM is significantly lower than that of conventional Nyquist sampled systems. In fact, around the frequency of interest (1kHz) the noise floor is 100dB below the signal



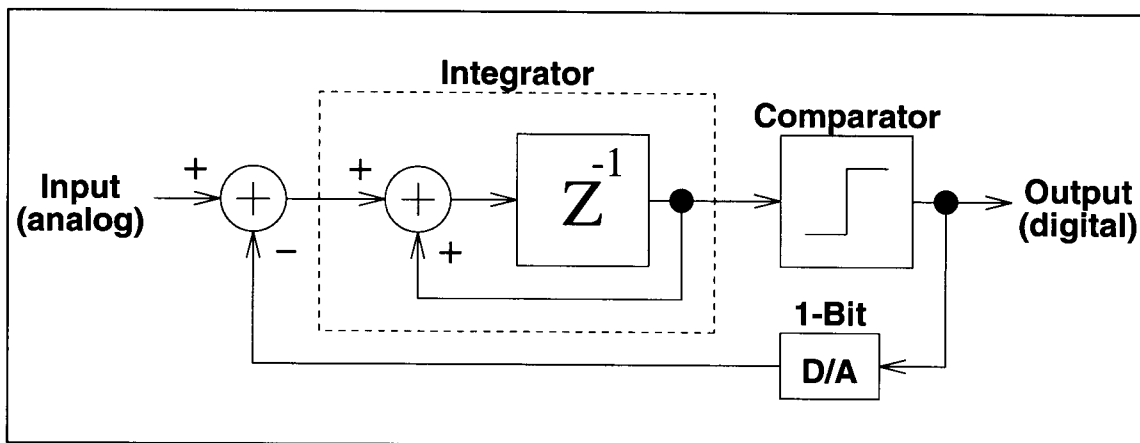


Figure 6: Block Diagram of a Delta-Sigma Modulator

power level, but as the frequency approaches Nyquist (500kHz) the noise floor levels off at around 20dB below the signal. It is this noise shaping that gives DSM such good performance at low frequencies. For a given frequency of interest, increasing  $R$  increases the signal-to-noise ratio (SNR). The usefulness of DSM is usually limited by the maximum frequency of interest and by the maximum attainable  $R$ . A mathematical model for the DSM is described in the next section.

### 2.2.3 Mathematical Model of DSM

The mathematical model of the DSM can be developed by assuming that the comparator in the DSM block diagram (figure 6) can be replaced with a white-noise source to represent the quantization noise. It is a safe assumption that the quantization noise resulting from the comparison is uniformly distributed over frequency, although this is a safer assumption for DSMs of order 2 or higher. First-order DSMs are prone to limit-cycles [2] and can therefore perform worse than theoretical predictions. Now assuming the comparator injects uniformly distributed quantization noise, we can replace it with an ideal noise source and determine the Z-domain transfer function for the system. Referring to figure 10 we have redrawn the system-level block diagram of the DSM, replacing the comparator with an ideal noise source  $E$  and replacing the integrator with the unit-delay Z-domain

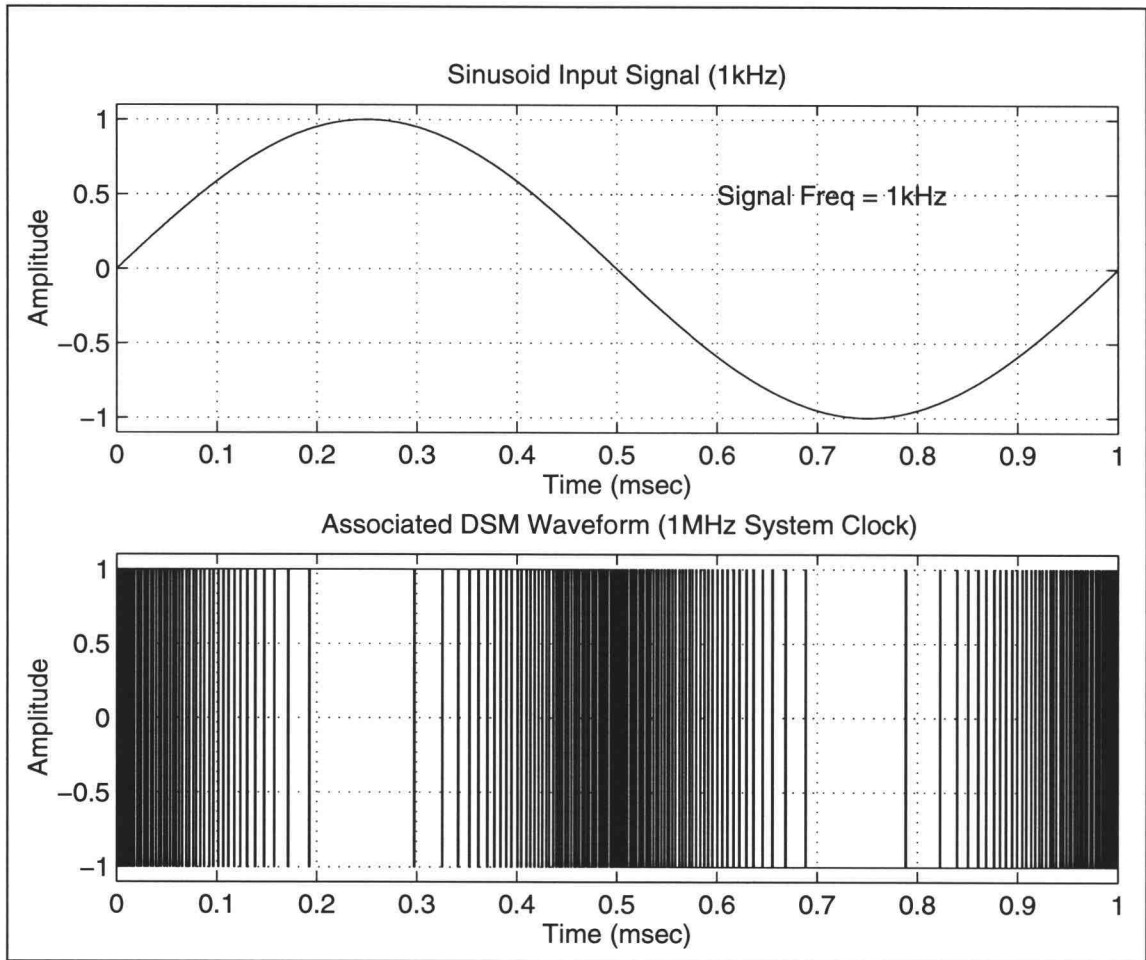


Figure 7: DSM Typical Waveform

equivalent. The resulting equation for the output signal is a sum of the original input signal plus a high-pass filtered version of the noise. This high-pass filtering is the noise-shaping mechanism described in the previous section. At low frequency the output closely approximates the input signal, but at high frequency (half the sampling frequency) the noise component is maximum. Alternatively, we could have used a zero-delay integrator in the system shown in figure 10, but to get the same transfer function would have required a unit delay in the feedback. Both methods are valid, but one may prove better than the other in a certain situation based on implementation issues. This model is fully described in reference [2].

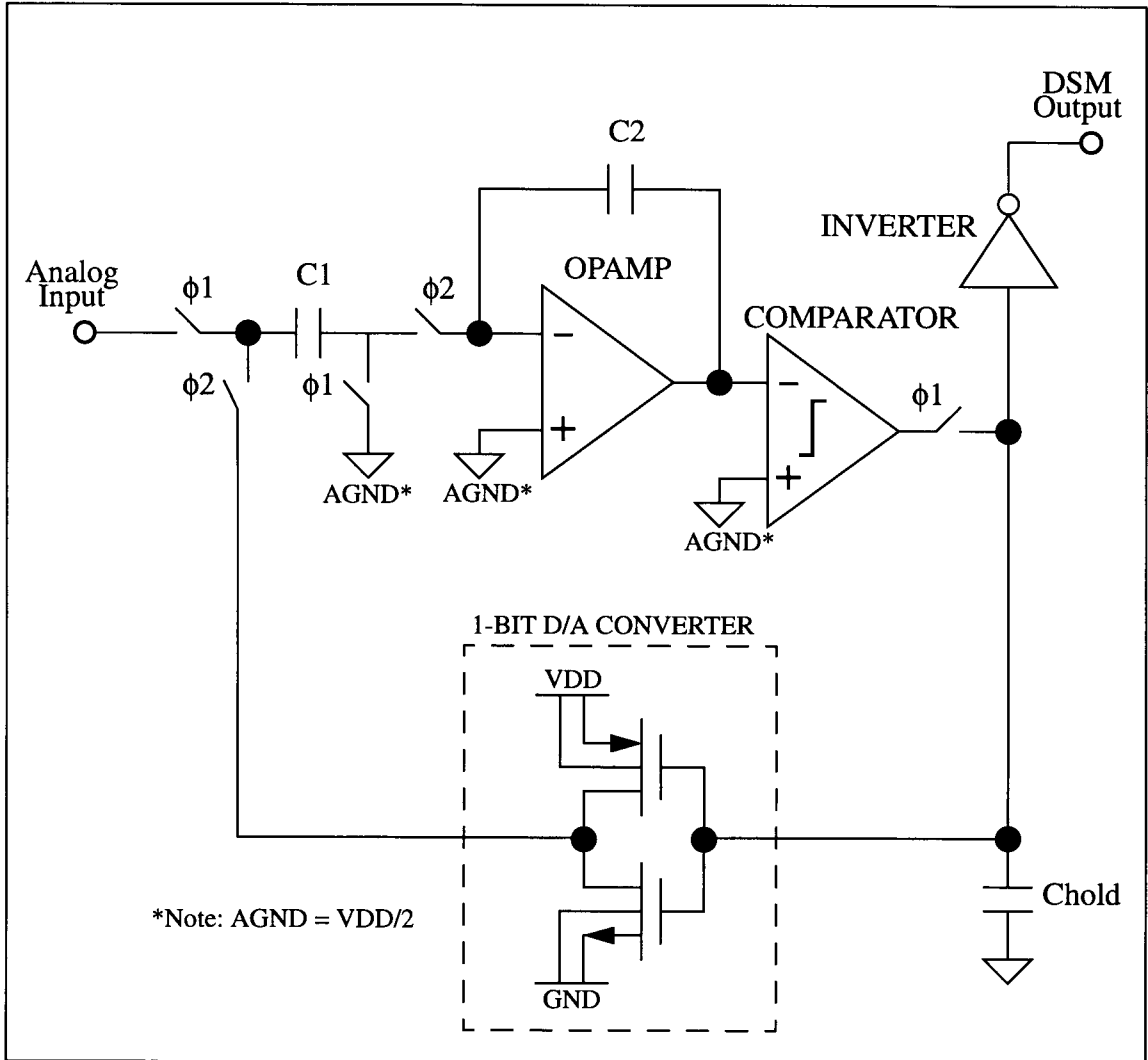


Figure 8: Switched-Capacitor Delta-Sigma Modulator (first order)

## 2.2.4 Applications

DSM is used extensively in audio-band circuits, and wherever an acceptable  $R$  can be attained for a desired input signal frequency. Audio is a well-behaved application since the upper frequency of interest is on the order of 20kHz. This allows the use of a relatively slow 20MHz system clock while still achieving an  $R$  value of 500. If a lower  $R$  is acceptable, say 100, then a 100kHz input signal could be modulated with a 20MHz clock. Exact SNR calculations for DSM output can be found in reference [2].

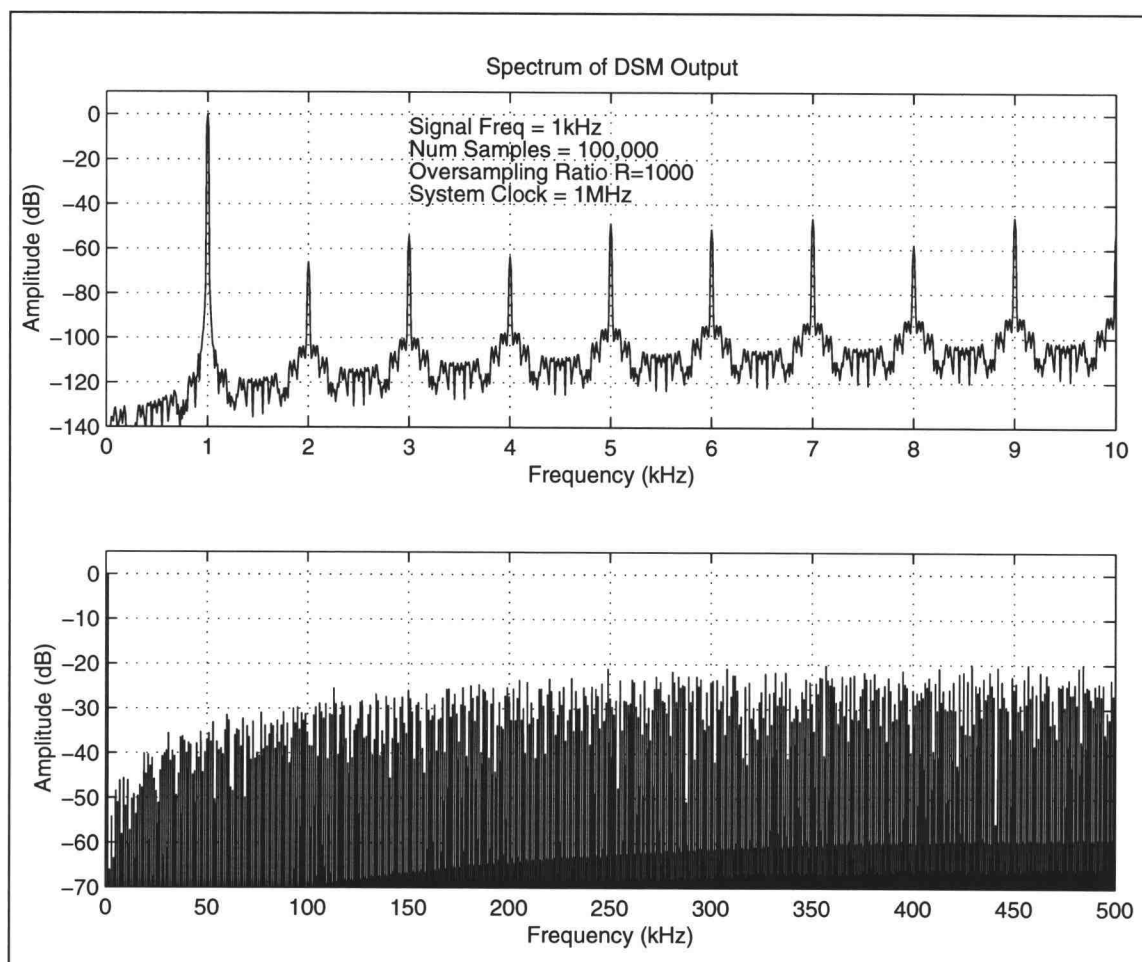


Figure 9: Noise Frequency Spectrum of Simulated DSM

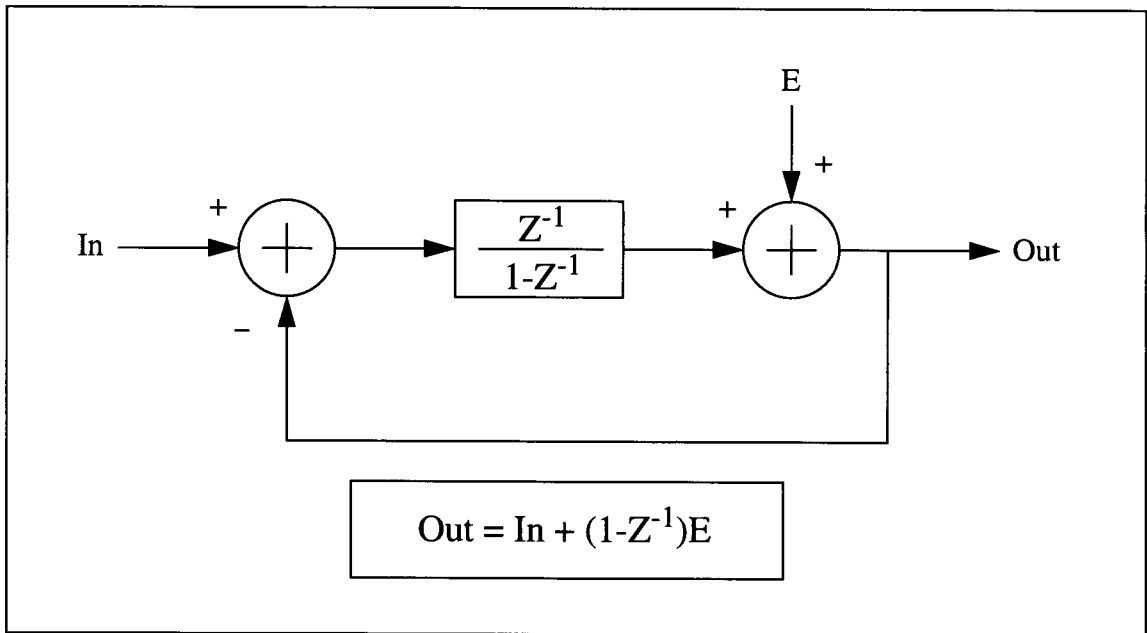


Figure 10: Z-Domain Equivalent Block Diagram of the DSM

### 3 DELTA-SIGMA PULSE WIDTH MODULATION (DSPWM)

The Delta-Sigma Pulse Width Modulator (DSPWM) is a hybrid circuit using DSM as an input stage and a digital PWM waveform generator as an output stage. This is illustrated below in figure 11. The DSM is a conventional first-order switched-capacitor implemen-

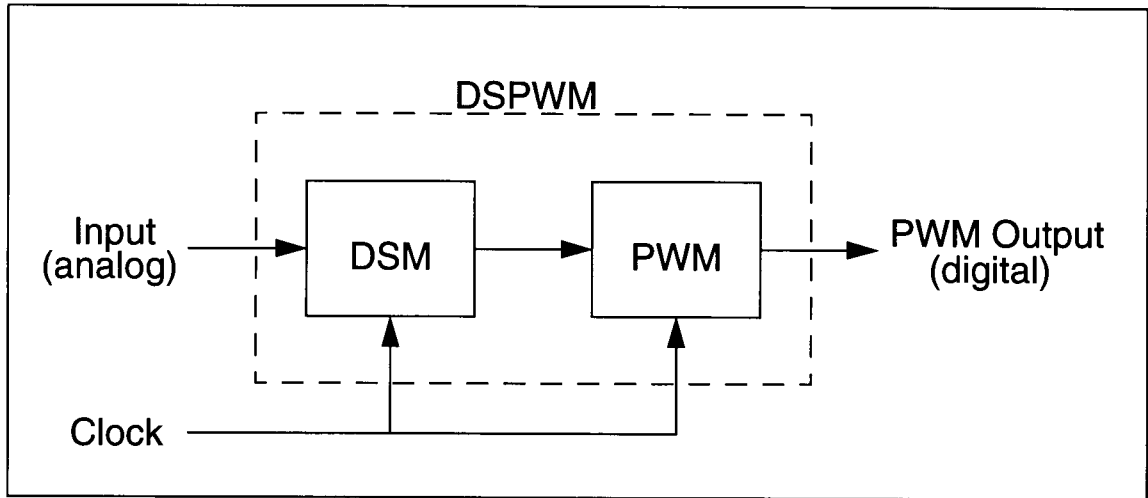


Figure 11: DSPWM Block Diagram

tation that inputs an analog waveform and outputs a digital bit stream. This digital bit stream feeds a digital PWM output stage which collects the DSM through a low-pass decimation filter (accumulate and dump) and reformats it into a PWM waveform. This PWM waveform has two characteristics which separates it from a conventional PWM waveform. First of all, the PWM pulses are dithered in time by a pseudorandom offset generator in the output stage. Secondly, the PWM pulses are selectively “inverted” and become active LOW going pulses when the effective PWM pulse is greater than 50% of the available PWM period. This later characteristic maximizes the effect of the dithering mechanism. The details of the DSPWM are illustrated in figure 12 below.

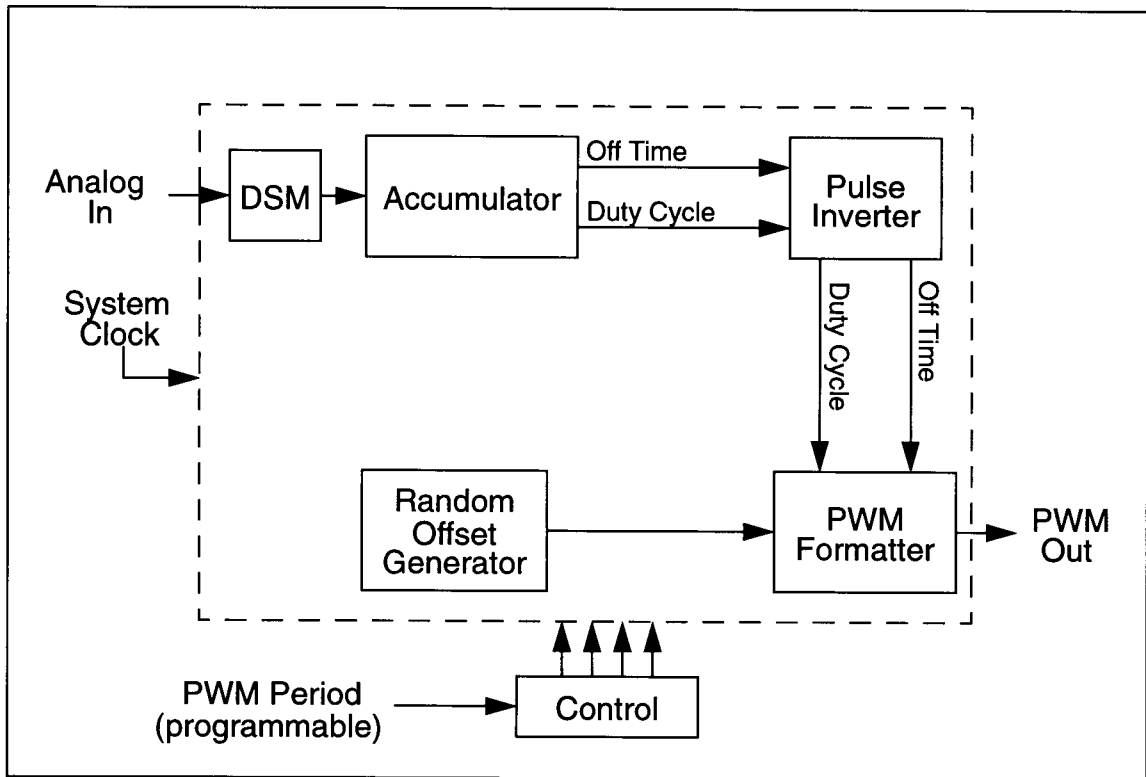


Figure 12: DSPWM Detailed Block Diagram

### 3.1 DSPWM Block Diagram

The individual blocks of the DSPWM are described in the following subsections.

#### 3.1.1 DSM

The DSM inside the DSPWM is a conventional first-order switched-capacitor implementation as described in section 2.2 above.

#### 3.1.2 Accumulator

The accumulator block integrates the incoming DSP bit stream over the defined PWM period and creates two parameters that are registered at the end of the current PWM

period for use down stream. The first parameter is the pulse duty cycle, or more appropriately, the effective PWM pulse width (in number of system clock cycles) that would represent the integrated DSM output over the PWM period. The second parameter is the pulse off-time (in number of system clock cycles) which represent the amount of time left in the PWM period beyond the PWM pulse. The sum of the two parameters “duty-cycle” and “off-time” always equals the total PWM period. An equivalent schematic of the accumulator is shown below in figure 13. This block is implemented as an 11-bit counter that is increased by one on each clock edge when the associated DSM output is in the high state (digital “1”). At the end of the current PWM period, the accumulated value is registered on the next rising edge of the clock, and the counter is either cleared or set to a value of “1” depending on the value of the associated DSM output. This method allows the registering of the accumulated data from the previous PWM period to occur while clearing the counter and accumulating for the first clock cycle of the next PWM period.

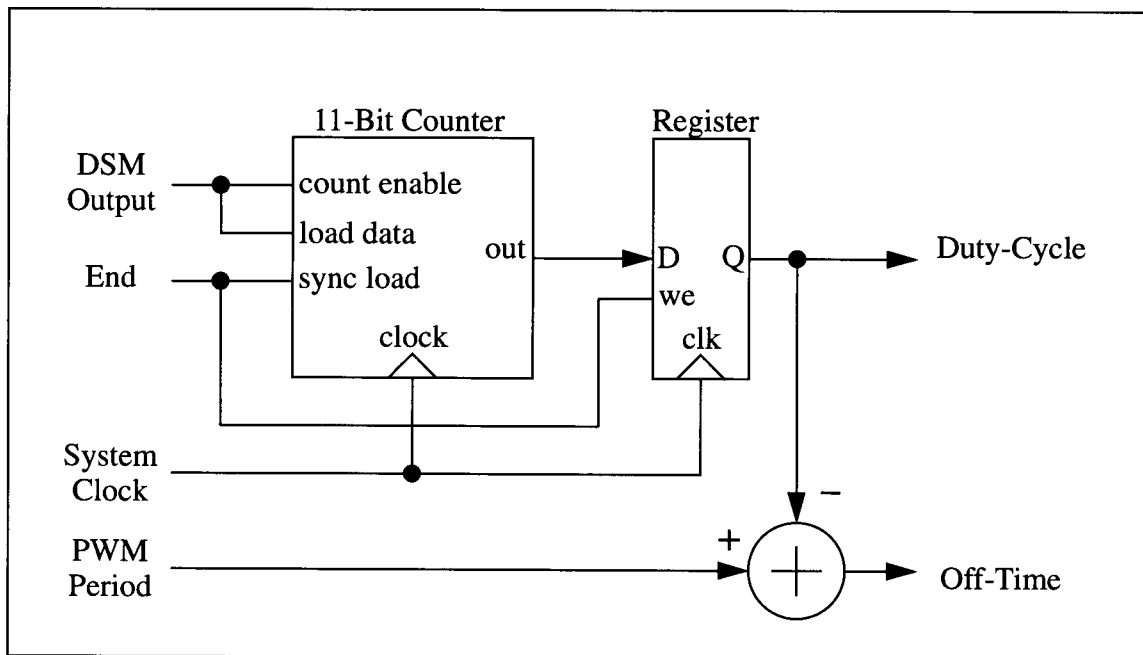


Figure 13: Accumulator Schematic Diagram



The accumulator acts as an N-tap FIR decimation filter with a  $\sin(x)/x$  frequency response, where N (decimation factor) is determined by the ratio of the system clock to the PWM switching frequency. The DSPWM has a programmable N value in the range of 127 to 2047. The  $\sin(x)/x$  response of this decimation filter has low-pass filter characteristics with the first zero of the frequency spectrum at the system clock frequency divided by N, or equivalently, the PWM frequency.

### 3.1.3 Pulse Inverter

The pulse inverter takes the two parameters from the Accumulator block and determines whether the duty-cycle parameter represents a value greater than or less than the total PWM period. Normally the PWM pulse is considered an active HIGH signal, where the PWM output is HIGH from the beginning of the PWM period until the end of the pulse, at which time the PWM output goes LOW. When the pulse inverter detects a duty-cycle parameter greater than 50% of the total PWM period, the pulse is converted from an active HIGH pulse to an active LOW pulse. This results in the two parameters “duty-cycle” and “off-time” switching values. So when inverted, the old “duty-cycle” becomes the new “off-time” and the old “off-time” becomes the new “duty-cycle”. This guarantees that the amount of available dither will be at least half of the PWM period for any given PWM pulse. If the input duty-cycle parameter is less than 50% of the PWM period, the “duty-cycle” and “off-time” are passed through the pulse inverter block unchanged. This procedure is described in more detail in section 3.2 below. An equivalent schematic diagram of the pulse-inverter is shown in figure 14.

### 3.1.4 PWM Formatter

The PWM formatter takes the final parameters “duty-cycle” and “off-time” from the pulse inverter and generates a PWM waveform. The pulse inverter sends a signal to the PWM formatter indicating whether the pulse is inverted or not. The PWM formatter takes the final off-time, increments it by one, multiplies it with a pseudorandom 11-bit fixed-point fraction, then truncates it to an integer. This creates an integer in the range

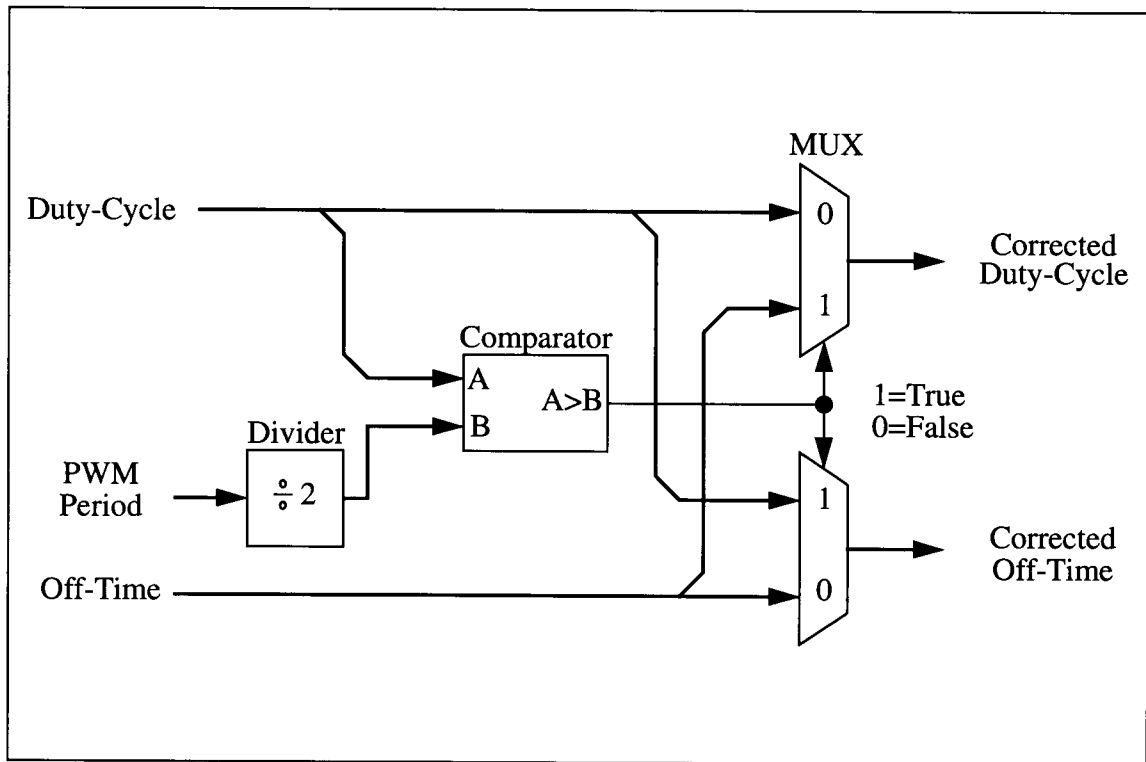


Figure 14: Schematic Diagram of the Pulse Inverter

[0,off\_time], where both endpoints 0 and “off-time” and all points in between have equal probability of being generated. The result of this multiplication is an offset that has a pseudorandom value in the range of [0,off-time] inclusive. This offset is used to delay the start of the PWM pulse. At the start of the new PWM period, a value equal to the OFF state is output for “offset” clock cycles. If the pulse is inverted, this is a HIGH, otherwise the OFF state is a LOW value. At the end of the offset, the PWM waveform goes to the ON state for “duty-cycle” clock cycles. After “duty-cycle” clock cycles the PWM waveform returns to the OFF state for the remainder of the PWM period. Details of this PWM waveform are discussed in section 3.2 below.

### 3.1.5 Random Offset Generator

The pseudorandom number generator is a 11-bit linear feedback shift register (LFSR) implementation. This consists of an 11-bit shift register using a specific pattern of bit taps that are XORed together (binary summed) and fed into the input of the shift register. A special circuit is used to detect a specific 11-bit (terminal) pattern that forces the LFSR into the all-zero state. The all-zero-state then maps to a specific 11-bit (initial) pattern. Normally a LFSR without this special circuit needs a non-zero pattern to be loaded into the registers. Since an all-zero pattern maps to an all-zero pattern, this is a lost code word, resulting in only  $2^{11}-1$  possible bit patterns. Using this special circuit to map to and from the all-zero state results in  $2^{11}$  possible bit patterns. It should also be made clear that the specific bit tap pattern chosen results in a “maximal” number of possible bit patterns for this 11-bit LFSR. A “non-maximal” implementation would result in a repeated pattern that contains less than  $2^{11}$  bit patterns. A schematic diagram on the Random Offset Generator is shown in figure 15 below.

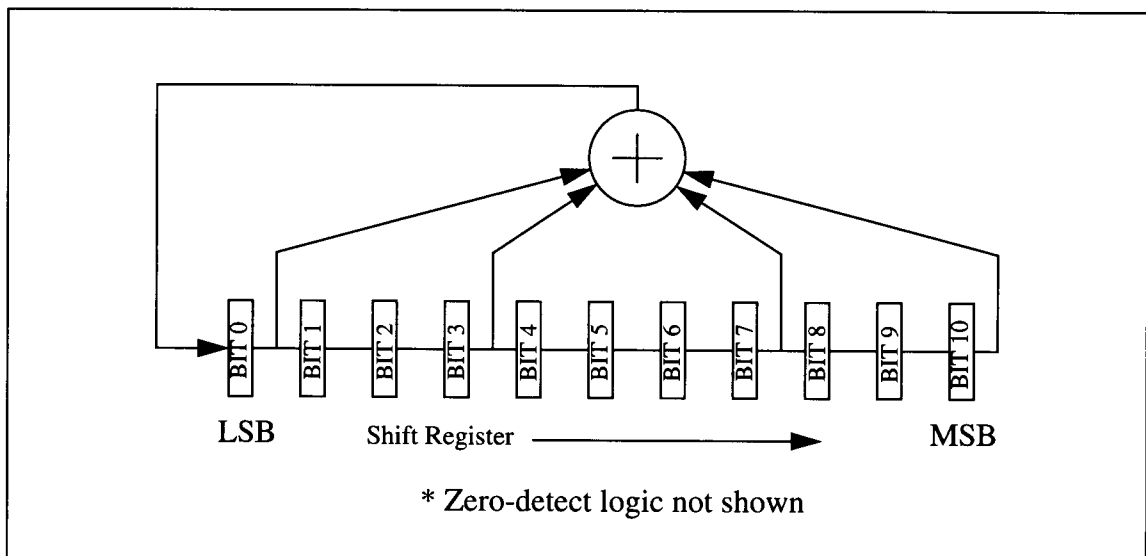


Figure 15: Schematic Diagram of the Random Offset Generator

### 3.2 Pulse Dithering and Inverting

The “dithering” and “inverting” characteristics are illustrated with a typical waveform in figure 16. Note that waveform (a) represents the PWM waveform as it comes from the accumulator block. Here the parameters “duty-cycle” and “off-time” define the PWM pulse for the previously integrated value of the DSM output over the last PWM period. Notice that the waveform on the left is less than 50% of the PWM period, while the waveform on the right is greater than 50% of the PWM period. Waveform (b) represents the PWM waveform after it is processed by the pulse inverter block. Notice that the waveform on the left (less than 50%) is unchanged as it passes through the pulse inverter, but the waveform on the right (greater than 50%) gets inverted and the parameters “duty-cycle” and “off-time” reverse values. Finally waveform (c) represents the PWM wave-

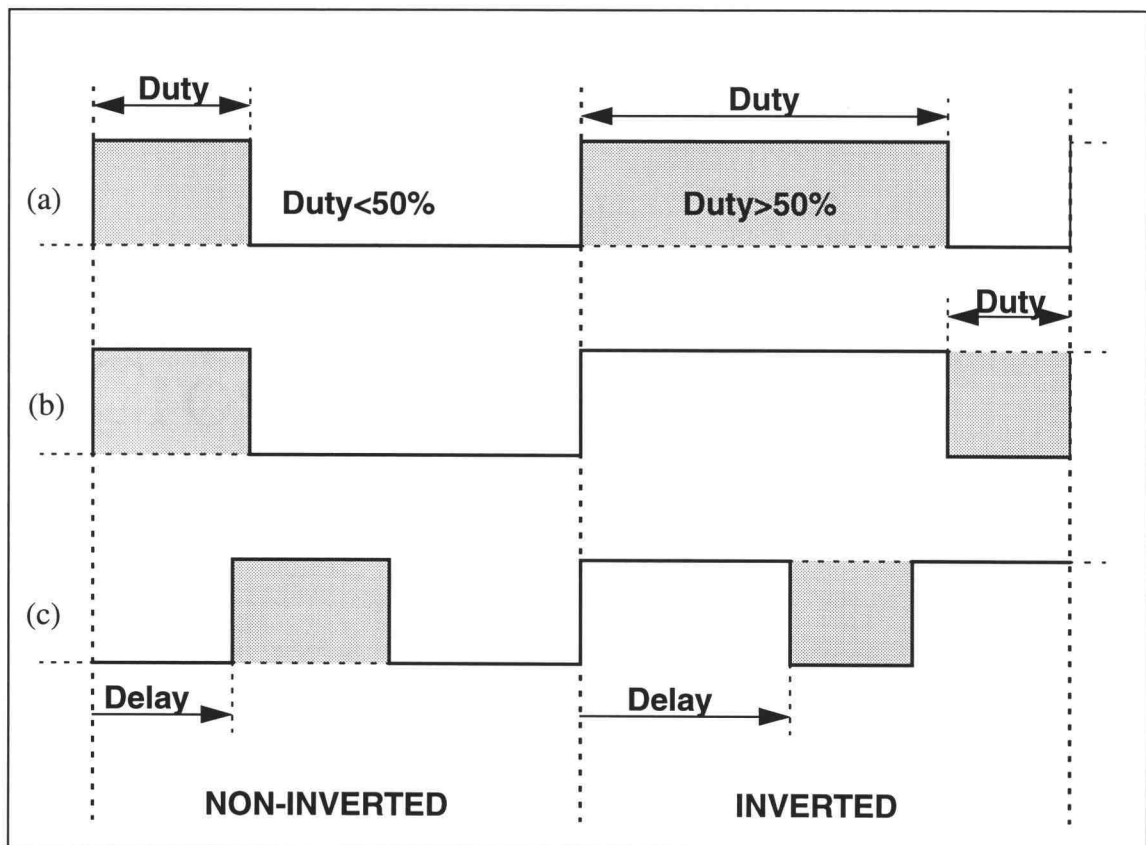


Figure 16: Dithered & Inverted Waveform Example

form from the PWM formatter (final output). The PWM formatter creates a PWM waveform that has a pulse width equal to “duty-cycle” system clock cycles, an offset equal to “offset” system clock cycles, and a polarity determined by the pulse inverter block (inverted or non-inverted). One unique characteristic of the dithering/inverting process is that when going from an “inverted” PWM pulse to a “non-inverted” PWM pulse from one PWM period to another, the random offset is disabled for that one PWM period (effectively zeroed out). This mechanism guarantees that the number of transitions on the DSPWM is exactly that of conventional PWM except in the case of the pseudorandom number being exactly 1 or exactly 0. This condition occurs twice in  $2^{11}$  PWM periods.

The original proof of concept of the DSPWM was done using MATLAB. The DSPWM system was modeled at a high level without regard to implementation issues.

#### 4.1 Model

The MATLAB DSPWM is modelled in two parts. The first part is an ideal DSM with a dynamic range of  $[-1, +1]$  volt. This DSM model dumps output to a data vector for temporary storage. This vector (when complete) is then run through a system-level implementation of the digital PWM output stage. The default single-precision of the HP9000/735 workstation was used for this simulation, and the digital values for the DSM output are defined to be -1 and +1. Though not ideal in a real single supply system, this choice of digital voltage levels makes calculations easy and removes the need to compensate for any DC offsets that would exist in a real system between real GND and virtual GND. The MATLAB simulation script files are included in appendix A.

#### 4.2 Simulation Results

The results of the MATLAB simulation are shown in figure 17 below. Figure 17 (a) shows the frequency spectrum of the input signal. In this case the input signal is a 1kHz sine wave of magnitude equal to 1 volt. Figure 17 (b) shows the spectrum of the DSPWM operating on the input, with no dithering. Figure 17 (c) shows the spectrum of the DSPWM operating on the input, with dithering enabled. For this simulation the input signal is a 1kHz sine wave of magnitude equal to 1 volt, the PWM switching frequency is 10kHz, and the system clock frequency is 1MHz. This results in an oversampling ratio of 500 for this particular input signal. The PWM fundamental period is equal to 100 system clock periods. The spectral plots are the result of taking 100,000 samples of the signal of interest, multiplying it by a Hanning window, then taking an FFT of the result. The frequency spectrum plot is the magnitude of the FFT. Note the significant reduction in harmonic noise peaks in the dithered DSPWM case compared to the non-dithered case.

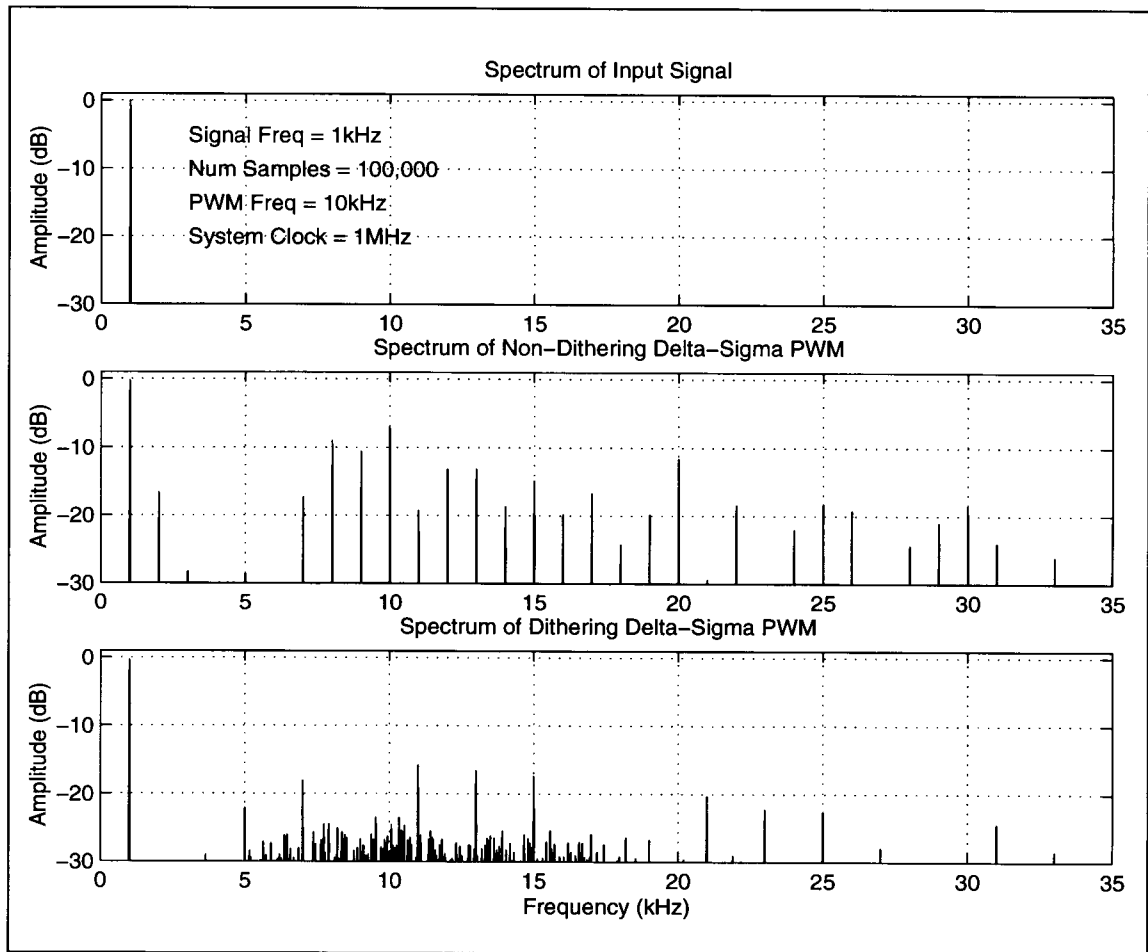


Figure 17: Matlab Simulation Results

## 5

## CIRCUIT IMPLEMENTATION

The implementation of the DSPWM consists of an analog block (switched capacitor DSM) and a digital block (digital PWM). A block diagram showing the system inputs and the analog/digital block interconnections is illustrated in figure 18 below. In this implementation there is a set of VDD/GND for the I/O pad drivers, a set for the analog block, and a set for the digital block. Separate power supplies helps to avoid on-chip noise coupling between the (noisy) I/O pads, the (noisy) digital logic, and the (sensitive) analog circuits. Externally, these supply pins can all be connected together to a 3.3 volts power supply. The SW[3:0] inputs are the upper four bits of the 11-bit constant that defines the PWM period in the digital block. With a binary range of between **00001111110** and **11111111110**, the PWM period can range from 6.4usec to 102.4usec when using a 20MHz system clock. The PWM period/frequency as a function of SW[3:0] is tabulated below in table 1. The reset input puts the digital logic into an initial state asynchronously. The dither\_enb signal enables dithering when HIGH. Dithering is disables when dither\_enb is LOW by forcing a pulse offset of zero in the digital PWM formatter block. The clock input is the system clock. This clock input also drives the 2-phase clock generator that generated the 2-phase clocks needed by the switched-capacitor DSM in the analog block. The analog inputs are Vin+, Vin-, and Vref. Vin+ is the analog input signal to be modulated. Vin- is currently unused, but will be used if the DSM gets converted to a fully differential DSM. The Vref input is a reference voltage input and should be tied to VDD/2 or 1.65 volts. The outputs of the DSPWM are PWM\_out, DSM\_out, and TPD. PWM\_out is the dithered PWM output from the entire system. The DSM\_out signal is the output of the DSM of the analog block. The TPD output is a MUXed version of the 2-phase clocks for use during debug to verify the health of the clocks to the analog block. The implementation of the DSPWM was done with different parameters than that of the proof-of-concept model. The reasoning for this was to allow for useful real-world input frequencies (audio) and still get a good R value in the DSM. The PWM period is programmable in this implementation to allow for characterizing the circuit at difference PWM frequencies.



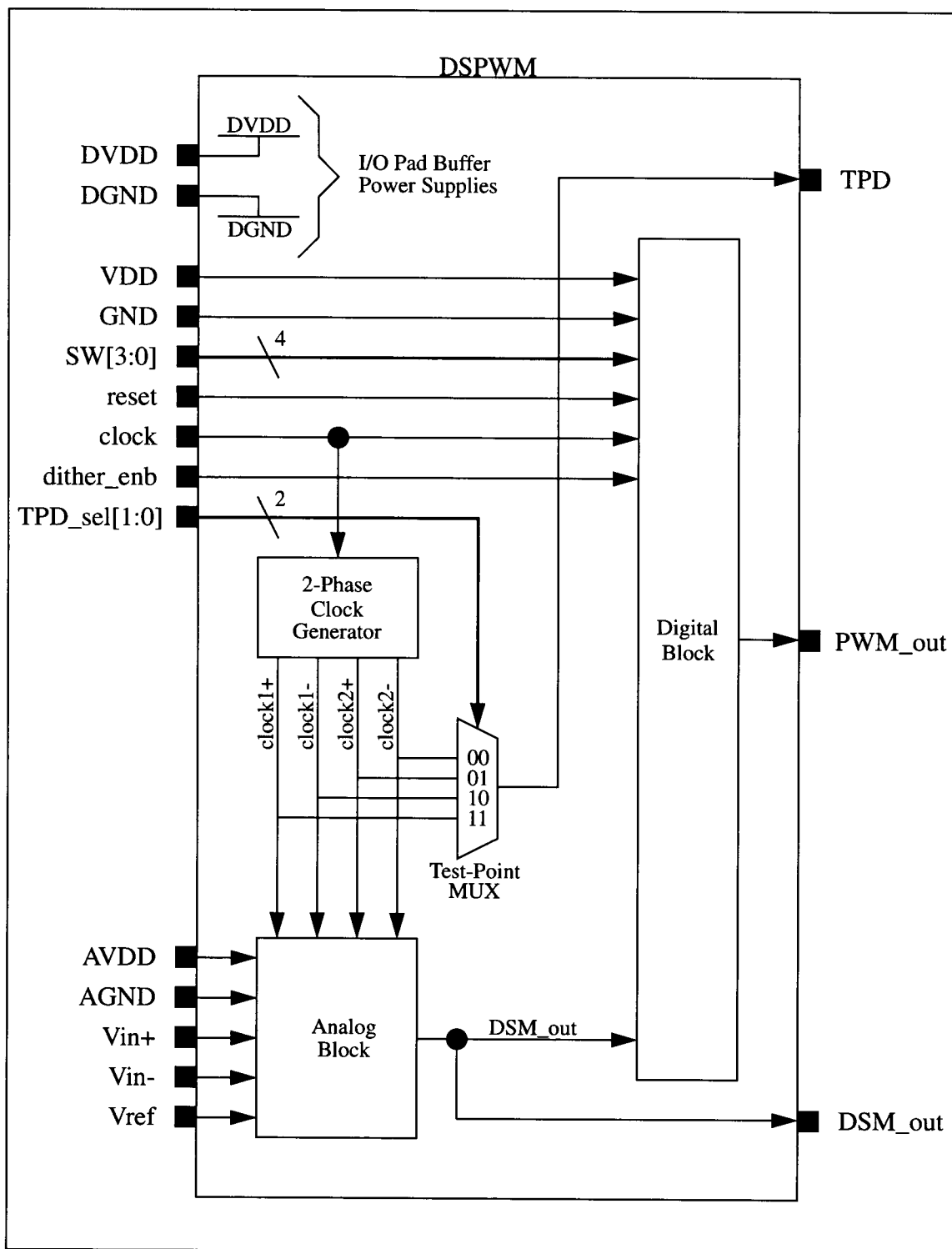


Figure 18: DSPWM System Inputs, Outputs, and Interconnects

SW[3:0] (binary)	PWM Period	PWM Frequency
0000	6.35 $\mu$ sec	157.480 kHz
0001	12.75 $\mu$ sec	78.431 kHz
0010	19.15 $\mu$ sec	52.219 kHz
0011	25.55 $\mu$ sec	39.139 kHz
0100	31.95 $\mu$ sec	31.299 kHz
0101	38.35 $\mu$ sec	26.076 kHz
0110	44.75 $\mu$ sec	22.346 kHz
0111	51.15 $\mu$ sec	19.550 kHz
1000	57.55 $\mu$ sec	17.376 kHz
1001	63.95 $\mu$ sec	15.637 kHz
1010	70.35 $\mu$ sec	14.215 kHz
1011	76.75 $\mu$ sec	13.029 kHz
1100	83.15 $\mu$ sec	12.026 kHz
1101	89.55 $\mu$ sec	11.167 kHz
1110	95.95 $\mu$ sec	10.422 kHz
1111	102.35 $\mu$ sec	9.770 kHz

Table 1: PWM Period and Frequency vs. SW[3:0] Using a 20MHz System Clock

## 5.1 Model

The DSPWM was modelled in two parts, an analog block for the switched-capacitor DSM and a digital block for the digital PWM output stage. The two blocks share a common system clock of 20MHz.

### 5.1.1 Analog Model

The analog model is the model for the DSM block. This block is modelled using HPSpice circuit simulator. The DSM analog model consists of several individual components and a top-level block that is the DSM. These blocks are described below

#### 5.1.1.1 DSM Top-Level Block

A schematic diagram of the top-level DSM block is shown below in figure 19. This is a first-order single-ended switched-capacitor DSM. This is the same as figure 8, except with capacitor values added. The capacitor ration of 1pF/2pF for C1/C2 allows the input voltage to traverse the entire analog dynamic range (0.0v-3.3v) while the output of the opamp will never attempt to exceed the bounds of the dynamic range. This avoids “clipping” of the integration value on C2. Using a capacitor ratio smaller than 2 could cause the integrator to saturate for certain inputs, resulting in distortion. The inputs to the block are VDD, GND, Vref, and Vin. VDD and GND are the power supply rails, Vref represents analog (virtual) GND and is ideally equal to VDD/2. Vin is the input signal to be operated on and eventually converted to PWM. The output of the block is the bit stream “DSM out” which feeds the accumulator of the digital PWM output stage. For our simulation purposes, ideal capacitors are used, but for more accurate analog simulation realistic capacitors need to be used. Unfortunately, there is no HPSpice model available for modelling a capacitor consisting of poly-silicon over gate oxide over highly doped (n++) substrate inside of a n-well, which is the method used in the HP 0.5 um process for creating linear capacitors in an analog process (AMOS14). This approximation does not adversely affect the validity of this “proof of concept”. Also, an ideal comparator was used in place of a real comparator. The ideal comparator is a native model in HPSpice. Also not shown is an ideal 100μA current source. This current source circuit has been designed, but it takes a significant amount of time to simulate the DSM with the current source in place. Again, this approximation does not take away from the validity of this “proof of concept”

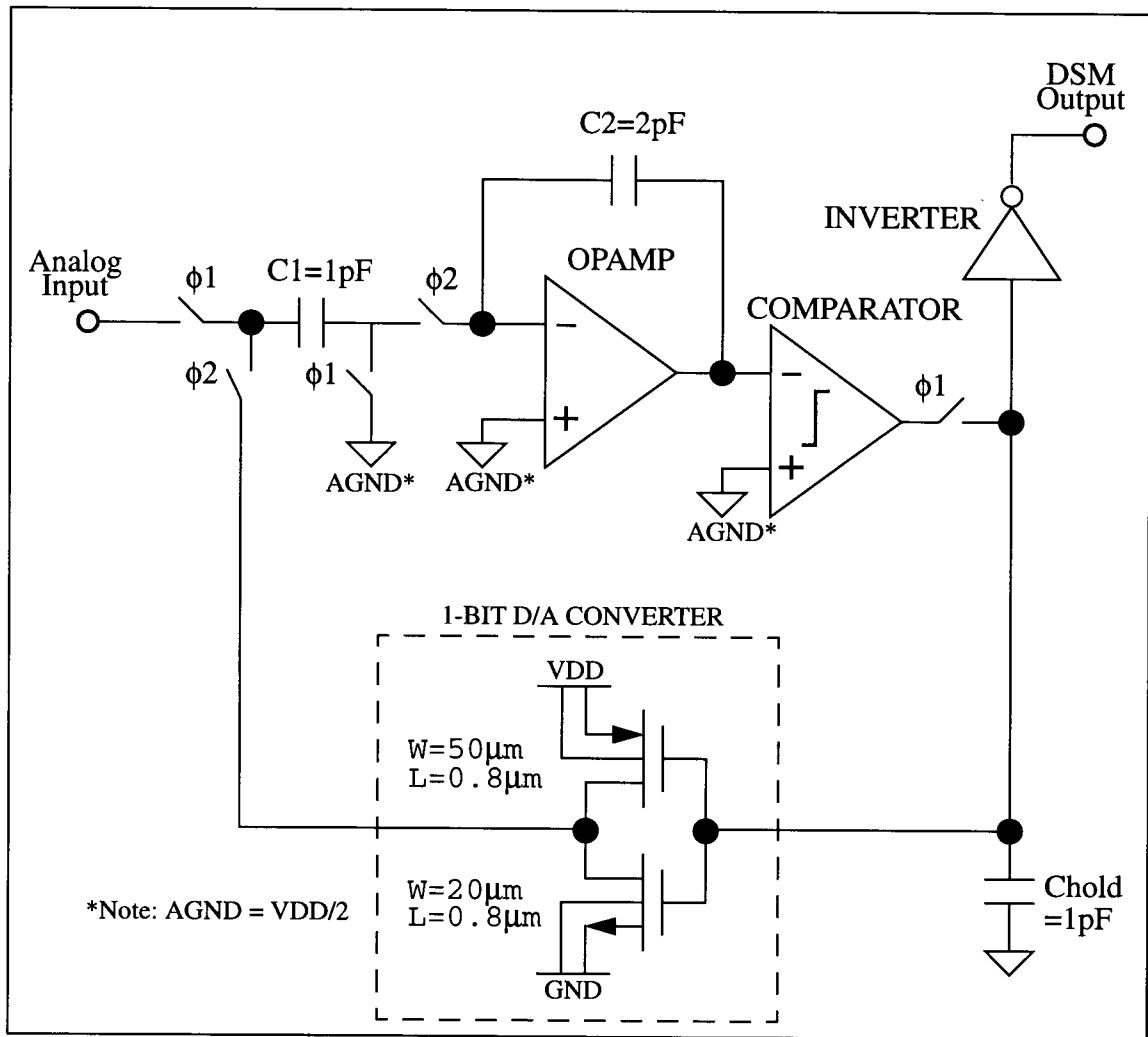


Figure 19: Schematic of DSM

### 5.1.1.2 OPAMP

The schematic diagram of the DSPWM opamp is shown in figure 20 below. The opamp used in the DSPWM was designed using a circuit optimizer called OPTSPICE. This is a Hewlett-Packard tool and cannot be fully described here. OPTSPICE is an optimizer built around the HPSpice simulator that uses the Simplex algorithm for optimizing a user defined cost function. The cost function consists of performance measurements for the circuit being optimized in the particular test bench used for optimization. OPTSPICE can

optimize device sizes, capacitances, and resistances based on desired AC, DC, and transient performance objectives.

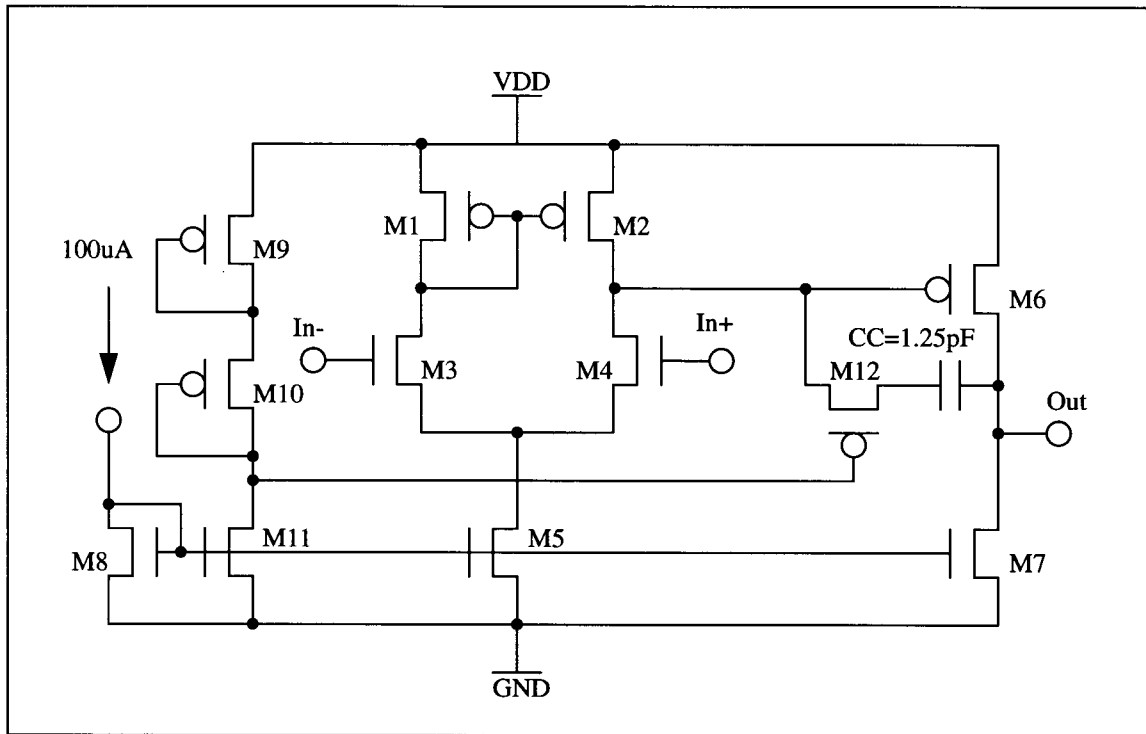


Figure 20: Schematic Diagram of Opamp

For the DSM circuit, initial device sizes were chosen to give reasonable performance. The optimizer was then used to get optimum performance. Several optimization objectives were defined to help the optimizer find a good solution. The objectives and the purpose for each objective is listed in table 2 below along with the final design performance. A “20dB-frequency” objective of 1 decade BELOW the unity-gain-frequency provides for a “straight line” approach to the 0db gain line. This proved to be a critical constraint in the optimization that forced a one-pole (20db/decade) gain roll-off as the gain crossed the 0dB line. This insured a good gain-margin. For simplicity a single gate length was used throughout the opamp. OPTSPICE was used to help evaluate the “best” perfor-

mance possible at several candidate gate lengths and it was determined that 0.8 $\mu$ m gate length provided the best results. The resultant device sizes are listed in table 3 below.

Objectives	Target Value	Attained Value
DC Gain	70 dB	66 dB
0dB Gain Freq (ugf)	100 MHz	128.8 MHz
20dbGain Freq	10 MHz	12.77 MHz
Phase Margin	70 degrees	70.0 degrees
Settling Time (0.1%) High	17 nsec	15.29 nsec
Settling Time (0.1%) Low	17 nsec	17.00 nsec
M6(Vdsat)	0.2 volts	0.197 volts
M7(Vdsat)	0.2 volts	0.193 volts
M1(Vds) - M2(Vds)	1.0 millivolts	0.992 millivolts

Table 2: Optimization Objectives for Opamp Design

	M1	M2	M3	M4	M5	M6
Width	109.3	109.3	16.0	16.0	49.5	1524.3
Length	0.8	0.8	0.8	0.8	0.8	0.8
	M7	M8	M9	M10	M11	M12
Width	349.3	8.9	1.8	1.8	8.9	22.7
Length	0.8	0.8	0.8	0.8	0.8	0.8

Table 3: Opamp Device Sizes (in  $\mu$ m)

### 5.1.1.3 Comparator

The use of a comparator is necessary in the final design, but the comparator was approximated in the simulation with an ideal comparator that switches at  $V_{DD}/2$ .

### 5.1.1.4 CMOS Switch

The schematic diagram of the CMOS switches used is shown in figure 21 below. This is a very conventional implementation. Reasonable values for the device sizes were chosen from simulation results. The devices were sized to give a fairly balanced charged/discharge time using a simple test circuit (10ns into 5pF). There are 4 extra devices in this switch to combat charge-injection. The extra devices are half the size of the main devices and are split among the output and the input. This architecture gives very good simulation results.

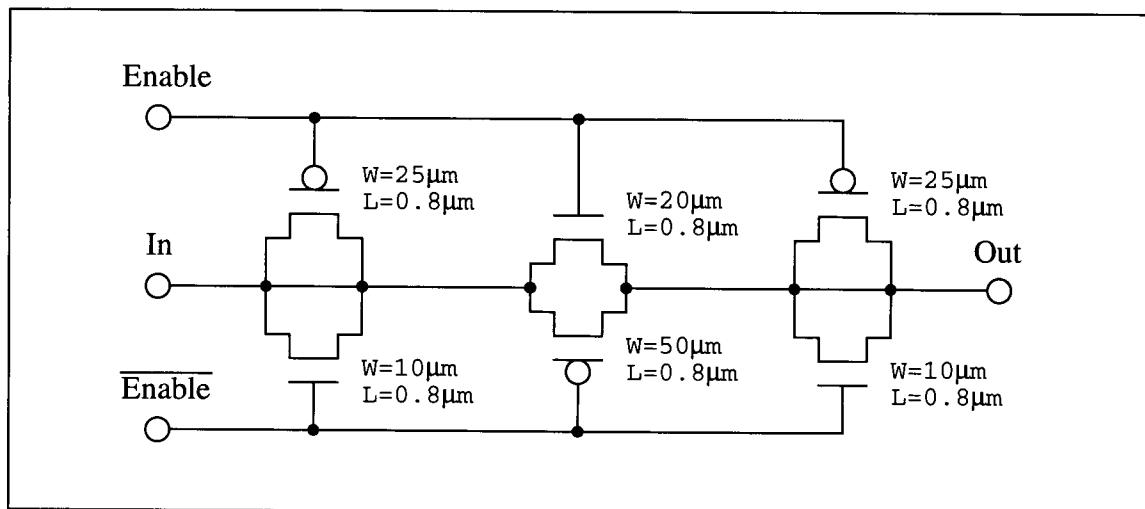


Figure 21: CMOS Switch Schematic Diagram

### 5.1.1.5 Current Source

The schematic diagram for the current source is shown in figure 22 below. This circuit is used to generate a 100uA reference current from Vref input. Vref should be carefully set to  $VDD/2 = 1.65$  volts externally, and the value of the external resistor should be carefully chosen to be 16500 ohms. This circuit is used in the opamp for generating bias voltages for various devices.

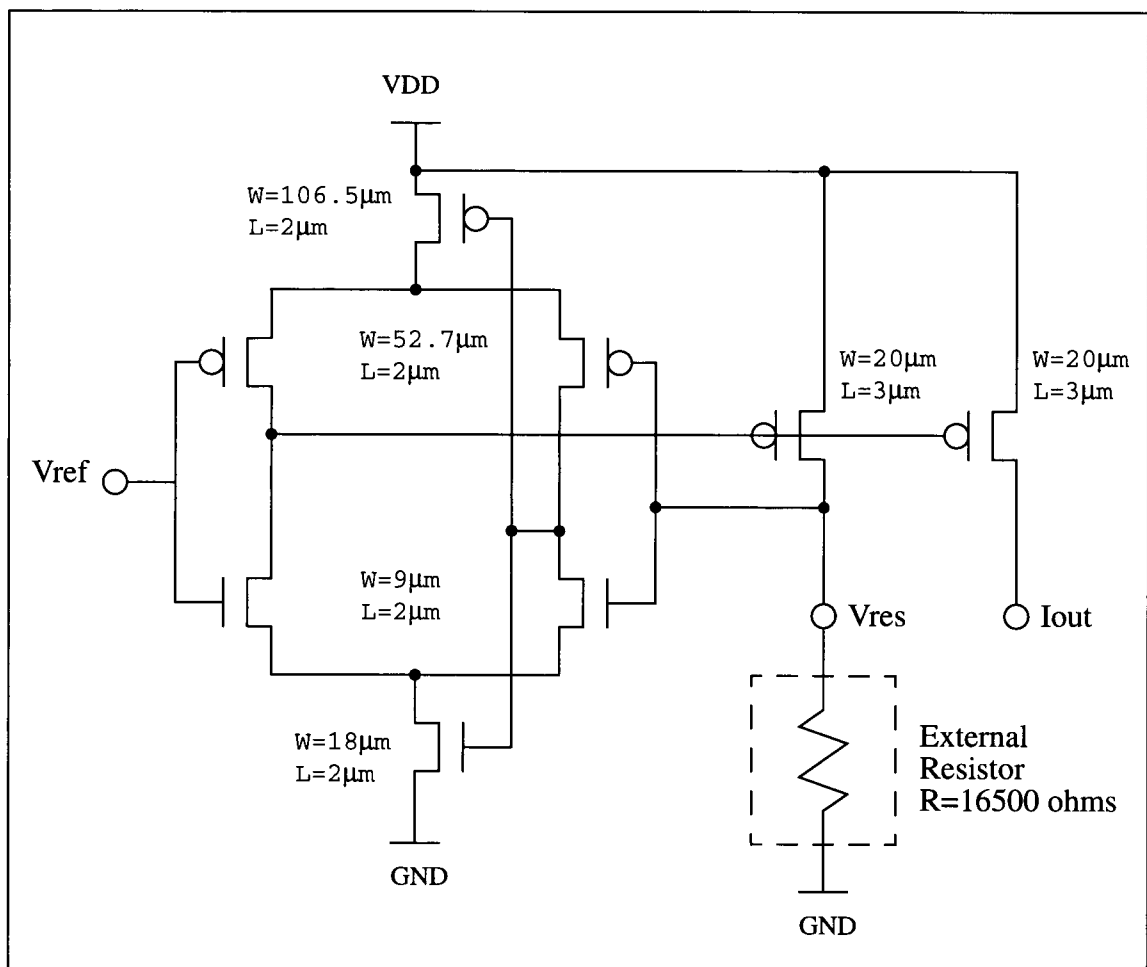


Figure 22: Current Source Schematic Diagram



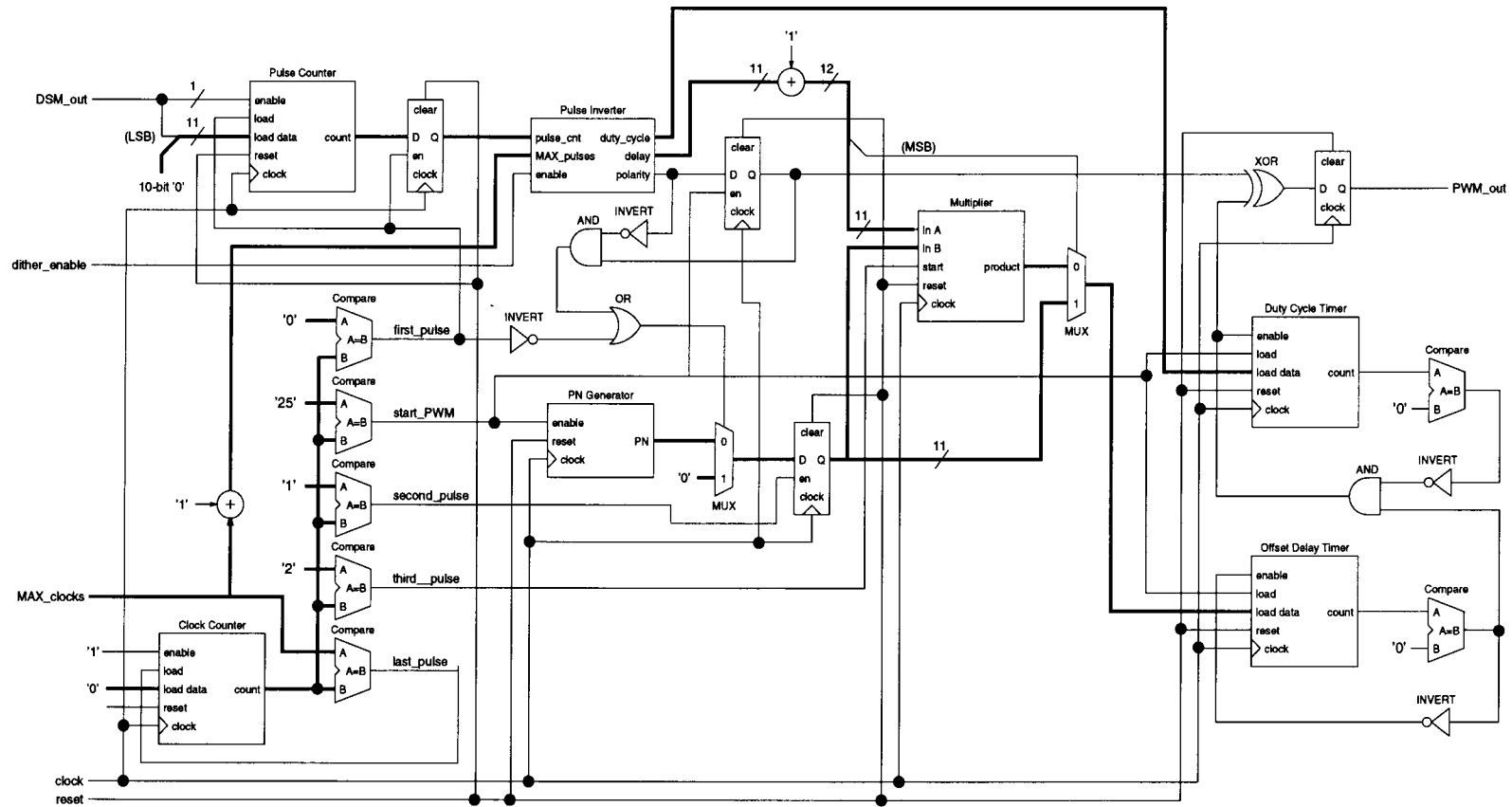
### 5.1.2 Digital Model

The digital model is used for the digital PWM output stage of the DSPWM. The schematic diagram of the PWM output stage is shown in figure 23 below. This is only representative of the PWM output stage functions, not the implementation. This model is completely digital and was implemented using Verilog Hardware Description Language (HDL) to describe the function of the digital circuits, and Synopsys synthesis tools to convert the HDL code to a gate-level netlist. The gate-level netlist maps the design to a specific set of standard cells within a specified standard cell library. In this case the Hewlett-Packard HPCMOS14 (0.5um) standard cell library version REV5\_0 was used. The Verilog HDL source code files are included in appendix B.

## 5.2 Simulation Results

The DSPWM was simulated in two parts. First the analog block was simulated in HPSpice and the resultant output bit stream was stored in a temporary file representing the DSM output. This DSM output was then formatted in MATLAB to be either a 1 or a 0 in value (simulating a comparator). This formatted output was then fed into the Verilog HDL simulation of the gate-level netlist of the DSPWM digital model. The output of the digital model was sampled during simulation and the samples were stored in a temporary file representing the DSPWM output. This output was read into MATLAB for processing and displaying. As before, the samples of the signal of interest were multiplied by a Hanning window and FFTed to get a frequency spectrum. For this circuit-level simulation the system clock was chosen to be 20MHz so that the PWM frequencies of table 1 could be used. The input switch settings for the PWM frequency were set to all zeros ( $SW[3:0] = 0000$ ) which results in a PWM frequency of 157.48kHz. The input signal was chosen to be a 4kHz sine wave to approximate a typical audio signal. Figure 24 shows the results of this simulation, where the upper graph in this figure is the frequency spectrum of a non-dithered PWM and the lower graph is the frequency spectrum of a dithered PWM. This simulation involved running two DSPWMs side-by-side with the same input data (from the DSM spice model), the same system clock, and the same switch settings

## Delta-Sigma Pulse Width Modulator (PWM Section)



Richard S. Lewison  
April 25, 1997

Figure 23: Schematic Diagram of Digital PWM Block

(SW[3:0]=0000), but with one having a “dither-enable” signal of 0 and the other having a “dither-enable” signal of 1. Notice that the clustering of the spectral noise at the PWM fundamental frequency is almost completely removed by dithering. There is some near-band noise that seems to be added, but the overall harmonic noise level is lower.

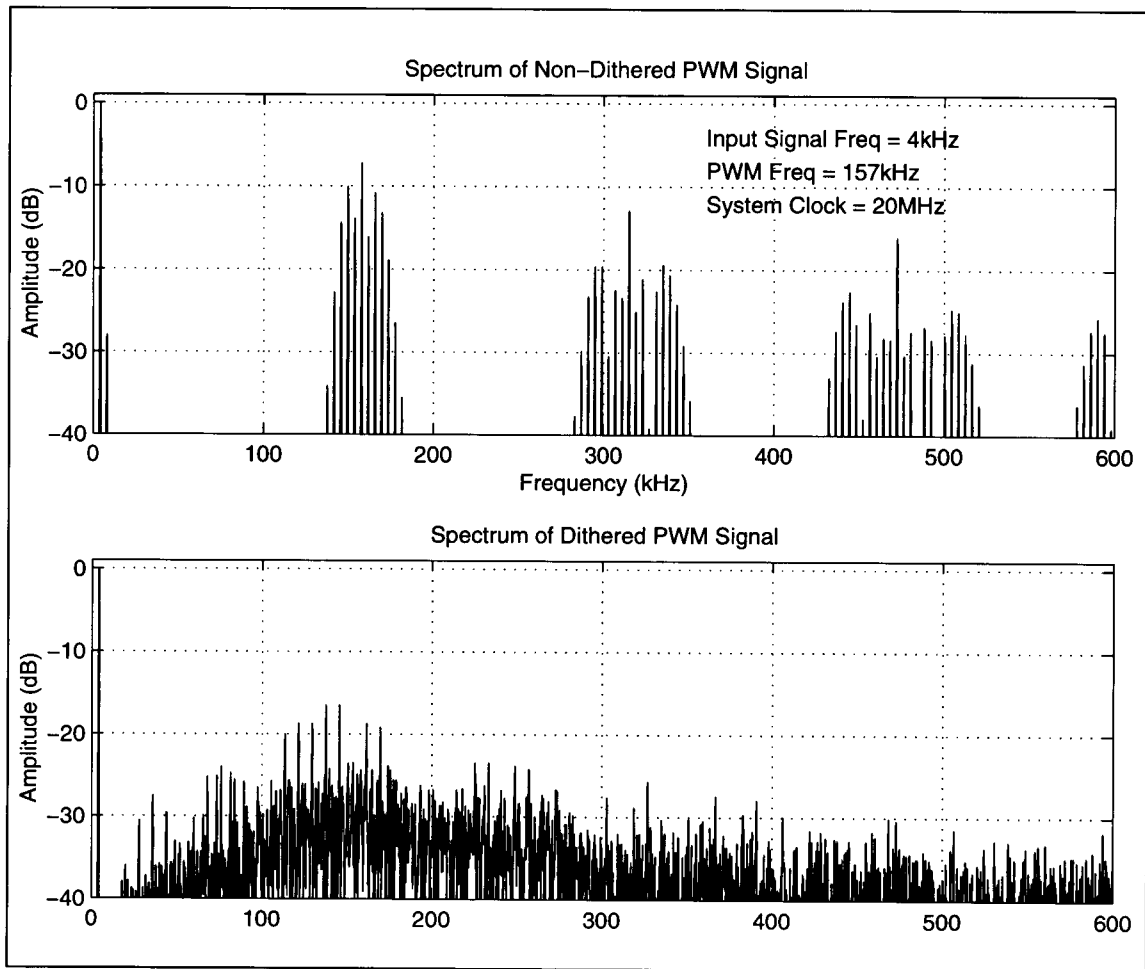


Figure 24: Spectral Plots of Simulation Using 4kHz Input signal

## 6

## CONCLUSION

Simulation has shown that DSPWM is a viable method for spreading spectral noise in systems requiring PWM. DSPWM also lends itself to be manufactured in a digital process since there is no requirement for precision resistors and capacitors, but only a requirement that capacitors are closely matched (for a switched-capacitor DSM only). Employing the DSPWM approach can reduce peaks in the frequency spectrum due to harmonic noise. From behavioral simulation and from circuit simulation it has been shown that the DSPWM can reduce harmonic noise energy peaks by at least 10 dB. This is a significant improvement in the frequency spectrum and requires no bulky filter components. The noise dithering is most effective when the PWM fundamental frequency is well above the input signal frequency. When a 4kHz input signal and a 157.48kHz PWM fundamental frequency was used in simulation, the noise peak reduction was greater than 10dB.

There is currently an effort to implement the DSPWM in the Hewlett-Packard CMOS14 (0.5um) digital process using a first-order switched-capacitor DSM and a fully digital standard-cell PWM dithering circuit. This circuit should be in silicon by late 1997.

## REFERENCES

- [1] N. Mohan, T. M. Undeland, W. P. Robbins., *Power Electronics: Converters, Applications, and Design*, 2nd ed., Wiley, 1995.
- [2] J. C. Candy et al., *Oversampling Delta-Sigma Data Converters: Theory, Design and Simulation*, New York, NY, IEEE Press, 1992.
- [3] Don Lancaster, "The Quest for Magic Sine Waves," *Circuit Cellar INK*, issue #59, pp. 28-35, June 1995.
- [4] T. Kato, "Precise PWM Waveform Analysis of Inverter for Selected Harmonic Elimination," 1986 IEEE/IAS Annual Meeting, pp. 611-616.
- [5] H. Patel and R. G. Hoft, "Generalized Techniques of Harmonic Elimination and Voltage Control in Thyristor Inverters: Part I - Harmonic Elimination," *IEEE Transactions on Industry Applications*, Vol. IA-9, No. 3, May/June 1973, pp.310-317.
- [6] J. Holtz, "Pulsewidth Modulation - A Survey," *IEEE Transactions on Industrial Electronics*, Vol. 39, No. 5, Dec. 1992, pp. 410-420.
- [7] R. S. Lewison and D. J. Allstot, "A Delta-Sigma Pulse Width Modulator with Pulse Dithering," 1997 Symposium on VLSI Circuits, June 1997, pp. 17-18.

## **APPENDICES**

## Appendix A    Matlab Source Code Files

```
% -----
% Matlab File to Implement conventional PWM.
%
%      By:      Richard S. Lewison
%      Date:    February 12, 1996
%      Mod:     June 3, 1997 (save environment upon exit)
% -----

% -----
% Define simulation parameters
% -----

clear;
simulation_time = .1;           % duration of simulation (sec)
sample_period = 1e-6;          % system clock = 1usec (1MHz)
Samples = fix(simulation_time/sample_period);
PWM_period = 1e-4;             % PWM period = 0.1msec (10kHz)
Max = fix(PWM_period/sample_period); % Number of clks in PWM period
signal_freq = 1000;            % Signal freq = 1kHz
period=round(PWM_period/sample_period); % PWM period (in clock ticks)

% -----
% Create time vector
% -----

time = [0:sample_period:simulation_time];

% -----
% Create Sawtooth Waveform
% -----

Saw = sawtooth(2*pi*time/PWM_period);

% -----
% Create Input Signal Waveform
% -----

Input = sin(2*pi*signal_freq*time);

% -----
% Generate conventional PWM from the Sigma-Delta output.
% -----

PWMout = sign(Input - Saw);      % Do PWM
PWMout = sign(PWMout + 0.1);     % get rid of zeros

% -----
% get spectrum of PWM signals
% -----

freq=(time/simulation_time)*(1/sample_period) - 0.5/sample_period;
Size = size(Input);
% Window = boxcar(Size(2));
% Window = hamming(Size(2));
Window = hanning(Size(2));
spect1=abs(fftshift(fft(Input.*Window')));
spect2=abs(fftshift(fft(PWMout.*Window')));
save DSPWM4;

if 1
```

```

Limit = 1.1*max([max(spect1) max(spect2)]);
max1=max(spect1);
subplot(2,1,1);
plot(freq/1000,20*log10(spect1/max1))
A = axis;
% axis([-2.5/PWM_period 2.5/PWM_period -30 1])
% axis([-0.0035/PWM_period .0035/PWM_period -30 1])
axis([0 .0035/PWM_period -30 1])
grid
title('Normalized Frequency Spectrum of Input Signal')
xlabel('Frequency (kHz)')
ylabel('Amplitude (dB)')
% Answer = sprintf('Signal Freq = %d Hz',signal_freq);
% text(0.0003/PWM_period,0.8*Limit,Answer);
% Answer = sprintf('Num Samples = %d',Samples);
% text(0.0003/PWM_period,0.7*Limit,Answer);
% Answer = sprintf('PWM Freq = %dkHz',fix(1/(1000*PWM_period)));
text(0.0003/PWM_period,0.6*Limit,Answer);
text(3,-5, 'Signal Freq = 1kHz')
text(3,-8, 'PWM Freq = 10kHz')
text(3,-11,'Num Samples = 100,000')
text(3,-14,'Sampling Rate = 1MHz')

subplot(2,1,2);
plot(freq/1000,20*log10(spect2/max1))
A = axis;
% axis([-2.5/PWM_period 2.5/PWM_period -30 1])
% axis([-0.0035/PWM_period .0035/PWM_period -30 1])
axis([0 .0035/PWM_period -30 1])
grid
title('Normalized Frequency Spectrum of Conventional PWM')
xlabel('Frequency (kHz)')
ylabel('Amplitude (dB)')

print -deps DSPWM4.eps

end

% -----
% Plot Input PWM output vs. time
% -----

if 0

subplot(2,1,1)
plot(time,Input,'-',time,Saw,'-')
axis([0 simulation_time -1.1 1.1])
title('PWM Input & Sawtooth Wave')
xlabel('Time (sec)')
ylabel('Amplitude (volts)')
grid

subplot(2,1,2)
plot(time,PWMout,'-')
axis([0 simulation_time -1.1 1.1])
title('PWM Output')
xlabel('Time (sec)')
ylabel('Amplitude (volts)')
grid

end

if 0

```



```

plot(time,Input,'-',time,PWMout,'-',time,Saw,'-')
axis([0 simulation_time -1.1 1.1])
title('PWM Input, Sawtooth Wave, & Output')
xlabel('Time (sec)')
ylabel('Amplitude (volts)')
grid

end

% -----
% Matlab File to Implement Delta-Sigma PWM using fixed
% PWM pulses (determined by signal) and random offsets
% from conventional PWM window. This version also
% splits up a pulse into a low-high-low pattern if the
% pulse count is less than 50% of MAXCOUNT, and splits
% up a pulse into a high-low-high pattern if the pulse
% count is greater than 50% of MAXCOUNT. This maximizes
% the jittering as the PWM duty cycle approaches unity.
% This version is the same as DSPWM5.m, except implemented
% more efficiently
%
% By: Richard S. Lewison
% Date: April 15, 1996
% Mod: January 14, 1997
% May 22, 1997 - to plot out DSM waveform
% June 39, 1997 - to plot the freq spect of DSM
% -----

% -----
% Define simulation parameters
% -----

clear;
simulation_time = .1; % duration of simulation (sec)
sample_period = 1e-6; % system clock = 1usec (1MHz)
PWM_period = 1e-4; % PWM period = 0.1msec (10kHz)
signal_freq = 1000; % Signal freq = 1kHz
period = round(PWM_period/sample_period); % number of clocks in PWM

% -----
% Create time vector
% -----

time = [0:sample_period:simulation_time];

% -----
% Create Sigma Delta bit stream
% -----

error=0.0; sigma=0.0;
fprintf(1,'\nCreating Sigma-Delta bit stream...');
for step = 1:simulation_time/sample_period + 1,
    t = time(step);
    signal(step) = sin(2*pi*signal_freq*t);
%    signal(step) = 0.8*sin(2*pi*signal_freq*t);
%    signal(step) = 0.5;
    sigma = sigma + error;
    out(step) = sign(sigma);
    out(step) = sign(out(step) + 0.1); % get rid of zeros
    error = signal(step) - out(step);
end

```

```

if 0
% -----
% plot DSM signal
% -----
subplot(2,1,1);
plot(time*1000,signal);
axis([0 1 -1.1 1.1]);
title('Sinusoid Input Signal (1kHz)')
ylabel('Amplitude')
xlabel('Time (msec)')
text(0.6, .5, 'Signal Freq = 1kHz')
grid;

subplot(2,1,2);
stairs(time*1000,out);
axis([0 1 -1.1 1.1]);
title('Associated DSM Waveform (1MHz System Clock)')
ylabel('Amplitude')
xlabel('Time (msec)')
grid;

print -deps DSPWM6a.eps

% -----
% get spectrum of DSM signal
% -----
fprintf(1, '\nGet spectrum of DSM stream...');
freq=(time/simulation_time)*(1/sample_period) - 0.5/sample_period;
window = hanning(length(time));
spect=abs(fftshift(fft(out.*window')));

% -----
% plot DSM spectrum
% -----
max0=max(spect);
Limit = 1.1*max0;

subplot(2,1,1);
plot(freq/1000,20*log10(spect/max0))
A = axis;
axis([0 10 -140 10])
grid
title('Spectrum of DSM Output')
ylabel('Amplitude (dB)')
xlabel('Frequency (kHz)')
text(3,-5, 'Signal Freq = 1kHz')
text(3,-15, 'Num Samples = 100,000')
text(3,-25, 'Oversampling Ratio R=1000')
text(3,-35, 'System Clock = 1MHz')

subplot(2,1,2);
plot(freq/1000,20*log10(spect/max0))
A = axis;
axis([0 500 -70 5])
grid
ylabel('Amplitude (dB)')
xlabel('Frequency (kHz)')

print -deps DSPWM6b.eps

end

exit

```

```

% -----
% Generate conventional PWM from the Sigma-Delta output.
% -----

count = 0; randcount = 0; downcount = 0;
step = 0; ustep = 0;
fprintf(1, '\nCreating conventional PWM bit stream...');
while step < fix(simulation_time/sample_period) + 1,

    if ustep == 0
        downcount = count;
        count = 0;
        randcount = 0;
    end;

    if (downcount > 0) & (randcount == 0)
        PWMout(step+1) = 1;
    else
        PWMout(step+1) = -1;
    end;

    count = count + (out(step+1)+1)/2;

    if randcount == 0
        downcount = downcount - 1;
        if downcount < 0, downcount = 0; end;
    end

    randcount = randcount - 1;
    if randcount < 0, randcount = 0; end;

    step = step + 1;
    ustep = rem(step,period);
end;

% -----
% Generate RANDOMIZED PWM from the Sigma-Delta output.
% -----

index = 1;

count = 0; randcount = 0; downcount = 0;
step = 0; ustep = 0;
fprintf(1, '\nCreating randomized PWM bit stream...');
while step < fix(simulation_time/sample_period) + 1,

    if ustep == 0
        if count <= fix(period/2)
            polarity = 1;
            downcount = count;
        else
            polarity = -1;
            downcount = period-count;
        end;
        randcount = round( rand*(period-downcount) );
        count = 0;
        test(index) = downcount + randcount;
        index = index + 1;
    end;

    if (downcount > 0) & (randcount == 0)
        PWMout(step+1) = 1*polarity;

```

```

else
    PWMROUT(step+1) = -1*polarity;
end;

count = count + (out(step+1)+1)/2;

if randcount == 0
    downcount = downcount - 1;
    if downcount < 0, downcount = 0; end;
end

randcount = randcount - 1;
if randcount < 0, randcount = 0; end;

step = step + 1;
ustep = rem(step,period);
end;

% -----
% get spectrum of PWM signals
% -----

fprintf(1,'\nGet spectrum of bit stream...');
freq=(time/simulation_time)*(1/sample_period) - 0.5/sample_period;
window = hanning(length(time));
spect1=abs(fftshift(fft(signal.*window')));
spect2=abs(fftshift(fft(PWMOUT.*window')));
spect3=abs(fftshift(fft(PWMROUT.*window')));

% -----
% plot signals
% -----

max1=max(spect1);
Limit = 1.1*max([max(spect1) max(spect2) max(spect3)]);

subplot(3,1,1);
plot(freq/1000,20*log10(spect1/max1))
A = axis;
axis([0 0.0035/PWM_period -30 1])
grid
title('Spectrum of Input Signal')
ylabel('Amplitude (dB)')
% text(0.0005/PWM_period,0.80*Limit,'Signal Freq = 1kHz')
% text(0.0005/PWM_period,0.65*Limit,'Num Samples = 100,000')
% text(0.0005/PWM_period,0.50*Limit,'PWM Freq = 10kHz')
% text(0.0005/PWM_period,0.35*Limit,'Sigma-Delta Freq = 1MHz')
text(3,-5,'Signal Freq = 1kHz')
text(3,-10,'Num Samples = 100,000')
text(3,-15,'PWM Freq = 10kHz')
text(3,-20,'System Clock = 1MHz')

subplot(3,1,2);
plot(freq/1000,20*log10(spect2/max1))
A = axis;
axis([0 0.0035/PWM_period -30 1])
grid
title('Spectrum of Non-Dithering Delta-Sigma PWM')
ylabel('Amplitude (dB)')

subplot(3,1,3);
plot(freq/1000,20*log10(spect3/max1))
A = axis;
axis([0 0.0035/PWM_period -30 1])

```

```

grid
title('Spectrum of Dithering Delta-Sigma PWM')
xlabel('Frequency (kHz)')
ylabel('Amplitude (dB)')

print -deps DSPWM6.eps
answer=input('Hit <RTN> to continue...');

% -----
% Plot PWM output vs. PWMR output
% -----

if 0
    subplot(2,1,1)
    plot(time*1000,PWMout,'-',time*1000,signal,'y:')
    title('Conventional PWM & Input')
    xlabel('Time (msec)')
    ylabel('Amplitude')
    axis([0 1 -1.1 1.1])
    grid

    subplot(2,1,2)
    plot(time*1000,PWMRout,'-',time*1000,signal,'y:')
    title('Jittered PWM & Input')
    xlabel('Time (msec)')
    ylabel('Amplitude')
    axis([0 1 -1.1 1.1])
    grid
end

fprintf(1,'\nDone!\n');

% -----
% Matlab File to Implement a pseudorandom number gener-
% ator. This file uses an N-bit linear shift register
% and an N-bit tap which is xored into the bit-1 posit-
% ion. Every possible tap is investigated for it's
% natural period. The goal is to find the OPTIMUM taps
% for a given N-bit linear shift register.
%
% By:      Richard S. Lewison
% Date:    April 25, 1996
% Mod:
% -----

% -----
% Define simulation parameters
% -----

clear
N = input('How Many Bits? '); % Length of Register
ans = input('Do you want output to a file [y/n] ? ','s');
if ans == 'y'
    fname = input('Output filename: ','s');
    outfile = fopen(fname,'w');
else
    outfile = 1;
end

% -----
% Begin Mask loop

```

```

% -----

fprintf(1,'\nWorking...');
fprintf(outfile,'\n\n');
for i = 1:(N/2-2),
    fprintf(outfile,' ');
end
fprintf(outfile,'Mask\tCount');
fprintf(outfile,'\n');
for i = 1:N,
    fprintf(outfile,'-');
end
fprintf(outfile,'\t-----');
fprintf(outfile,'\n');
for Mask_decimal = 1:2^N-1
    temp = Mask_decimal;
    for i = 1:N
        Mask(i) = fix(temp/2^(N-i));
        temp = temp - Mask(i)*2^(N-i);
    end
    % -----
    % Search for natural period
    % -----

    LSR = Mask; Start = 0; Count = 0; LSRlast = LSR;
    while ((sum(LSR ~= Mask) ~= 0) | (Start == 0)) & (Count < 2*2^N)
        LSRlast = LSR;
        Start = 1;
        bit1 = rem(sum(Mask & LSR),2);
        LSR(2:N) = LSR(1:N-1);
        LSR(1) = bit1;
        Count = Count + 1;
    end
    fprintf(outfile,'%d',Mask);
    if Count > (2*2^N)-1, Count = 1; end
    if Count == 2^N-1,
        fprintf(outfile,'\t %d *\n',Count);
    else
        fprintf(outfile,'\t %d\n',Count);
    end
end
fprintf(outfile,'\n');
fprintf(1,'...Done!\n\n');
fclose('all');

% -----
% This is a Matlab script file for taking data from the final
% verilog simulation of the DSPWM and plotting the spectrum.
%
%   File:   plot_spectrum.m
%   Date:   07/06/97
%   Author: Richard Lewison
%
% (1) input:  <user prompted filename of Verilog output data>
%             ...either pwmout_20kHz.dat or pwmout_4kHz.dat
%
% (2) output: plot_spectrum.eps - plot of frequency spectrum
%
% -----
% -----

```

```

% Input the data from the Verilog simulation
% -----
file=input('What is input filename containing Verilog output data: `', 's');
load(file);
time=pwmout(:,1)-pwmout(1,1);    % time vector
pwm1=pwmout(:,2)-0.5;            % dithered PWM
pwm2=pwmout(:,3)-0.5;            % non-dithered PWM

% -----
% get spectrum of PWM signals
% -----
fprintf(1, '\nGet spectrum of PWM signals...');
len=length(time);
simulation_time=time(len)-time(1);
sample_period=simulation_time/len;
freq=(time/simulation_time)*(1/sample_period) - 0.5/sample_period;
window = hanning(len);
spect1=abs(fftshift(fft(pwm1.*window)));
spect2=abs(fftshift(fft(pwm2.*window)));

% -----
% plot signals
% -----

max1=max(spect1);

if (strcmp(file, 'pwmout.20kHz.dat') == 1),    % for 20kHz signal

    subplot(2,1,1);
    plot(freq/1000, 20*log10(spect2/max1))
    axis([0 600 -40 1])
    grid
    title('Spectrum of Non-Dithered PWM Signal')
    ylabel('Amplitude (dB)')

    text(350,-3, 'Input Signal Freq = 20kHz')
    text(350,-7, 'PWM Freq = 157.48kHz')
    text(350,-11, 'System Clock = 20MHz')

    subplot(2,1,2);
    plot(freq/1000, 20*log10(spect1/max1))
    axis([0 600 -40 1])
    grid
    title('Spectrum of Dithered PWM Signal')
    ylabel('Amplitude (dB)')
    xlabel('Frequency (kHz)')

    print -deps plot_spectrum_20kHz.eps

end

if (strcmp(file, 'pwmout.4kHz.dat') == 1),    % for 4kHz signal

    subplot(2,1,1);
    plot(freq/1000, 20*log10(spect1/max1))
    axis([0 600 -40 1])
    grid
    title('Spectrum of Dithered PWM Signal')
    ylabel('Amplitude (dB)')

    text(300,-5, 'Input Signal Freq = 4kHz')
    text(300,-10, 'PWM Freq = 157kHz')
    text(300,-15, 'System Clock = 20MHz')

```

```
subplot(2,1,2);
plot(freq/1000,20*log10(spect2/max1))
axis([0 600 -40 1])
grid
title('Spectrum of Non-Dithered PWM Signal')
ylabel('Amplitude (dB)')
xlabel('Frequency (kHz)')

print -deps plot_spectrum_4kHz.eps

end

fprintf(1,'---Done---\n');
```



## Appendix B Verilog HDL Source Code Files

```

/*****
*                               Sigma-Delta Pulse-Width Modulator (SDPWM)
*
*   File       : DSPWM.v (TOP level file)
*   Module(s)  : DSPWM
*   Engineer   : Richard S. Lewison
*   Date       : February 13, 1997
*               February 18, 1997
*               February 28, 1997
*
* ----- Description -----
* This module is the top-level module for the Delta Sigma Pulse Width Modulator
* (DSPWM).
*****/
`timescale 1 ns / 10 ps

module DSPWM (
    clock,          // In - System clock input
    reset,          // In - Asynchronous reset input (active high)
    Vinp,           // InOut - Analog input signal (+)
    Vinn,           // InOut - Analog input signal (-)
    Vref,           // In - Analog ground (reference)
    dither_enb,     // In - Dither enable flag (0 = no dither)
    SW,             // In - 4-bit DIP switch setting for max[10:7]
    PWM_out,        // Out - DSPWM output (PWM waveform)
    TPD_sel,        // In - 2-bit selector for test point
    TPD,            // Out - Digital test point (for analog clocks)
    AVDD,           // InOut - Analog VDD
    AGND            // InOut - Analog GND
);

//-----
// Inputs & outputs
//-----

input [3:0] SW;
input [1:0] TPD_sel;
input dither_enb, reset, clock;
output PWM_out, TPD;
inout Vinp, Vinn, Vref, AVDD, AGND;

wire clock, clk1N, clk1P, clk2N, clk2P, TPD;
wire DSM_out, PWM_out, reset, dither_enb;
wire [3:0] SW;
tri Vinp, Vinn, Vref, AVDD, AGND;

//-----
// ASSIGN statements
//-----

assign TPD = ( (TPD_sel == 2'b00) & clk1P      // clk1P -> Test point
              | (TPD_sel == 2'b01) & clk1N      // clk1N -> Test point
              | (TPD_sel == 2'b10) & clk2P      // clk2P -> Test point
              | (TPD_sel == 2'b11) & clk2N );    // clk2N -> Test point

//-----
// Instantiate other modules
//-----

```

```

TOP_analog Analog (
    .Vinp(Vinp),          // InOut - Analog input signal (+)
    .Vinn(Vinn),          // InOut - Analog input signal (-)
    .Vref(Vref),          // In - Analog ground (reference)
    .clk1P(clk1P),        // Phase 1 (+) switch capacitor clock
    .clk1N(clk1N),        // Phase 1 (-) switch capacitor clock
    .clk2P(clk2P),        // Phase 2 (+) switch capacitor clock
    .clk2N(clk2N),        // Phase 2 (-) switch capacitor clock
    .reset(reset),        // Input --> asynchronous reset
    .DSM_out(DSM_out),    // Output -> modulator digital output
    .AVDD(AVDD),          // InOut - Analog VDD
    .AGND(AGND)           // InOut - Analog GND
);

aclk_gen Aclk_gen (
    .clock(clock),        // Input --> clock reference
    .clk1P(clk1P),        // Output -> Phase 1 clock (+)
    .clk1N(clk1N),        // Output -> Phase 1 clock (-)
    .clk2P(clk2P),        // Output -> Phase 2 clock (+)
    .clk2N(clk2N)         // Output -> Phase 2 clock (-)
);

TOP_digital Digital (
    .clock(clock),        // In - clock input from crystal oscillator
    .reset(reset),        // In - Asynchronous reset inpiu (active high)
    .dither_enb(dither_enb), // In - Dither enable flag (0 = no dither)
    .SW(SW),              // In - DIP switch setting for max[10:7]
    .DSM_out(DSM_out),    // In - output from Delta-Sigma modulator
    .PWM_out(PWM_out)     // Out - DSPWM output (PWM waveform)
);

endmodule

/*****
*                               Sigma-Delta Pulse-Width Modulator (SDPWM)                               *
*                               *                               *                               *
*   File      : aclk_gen.v                                           *
*   Module(s) : aclk_gen                                             *
*   Engineer  : Richard S. Lewison                                   *
*   Date      : February 28, 1997                                     *
*                               *                               *                               *
*   ----- Description ----- *
*   This module generates the two non-overlapping clocks for switched-capacitor *
*   circuits in TOP_analog.                                           *
*****/
`timescale 1 ns / 10 ps

module aclk_gen (
    clock,          // Input --> clock for counter
    clk1P,          // Output -> Phase 1 clock (+)
    clk1N,          // Output -> Phase 1 clock (-)
    clk2P,          // Output -> Phase 2 clock (+)
    clk2N           // Output -> Phase 2 clock (-)
);

//-----
// define inputs/outputs
//-----

input  clock;
output clk1P,clk1N,clk2P,clk2N;

reg clockN,clk1N,clk2N;

```

```

reg clk1N1,clk2N1;
wire clk1P = !clk1N;
wire clk2P = !clk2N;

//-----
// define asynchronous logic
//-----

always @ (clock)
begin
    #1 clockN = !clock;
end

always @ (clock or clk2N)
begin
    #3 clk1N1 = !(clock & clk2N);
end

always @ (clockN or clk1N)
begin
    #3 clk2N1 = !(clockN & clk1N);
end

always @ (clk2N1)
begin
    #3 clk2N = clk2N1;
end

always @ (clk1N1)
begin
    #3 clk1N = clk1N1;
end

endmodule

/*****
*                               Sigma-Delta Pulse-Width Modulator (SDPWM)
*                               *
*   File       : counter11.v
*   Module(s)  : counter11
*   Engineer   : Richard S. Lewison
*   Date       : February 1, 1997
*               February 9, 1997
*
* ----- Description -----
* This module is a generic 11-bit up/down loadable counter with count enable
* and synchronous reset.
*****/
`timescale 1 ns / 10 ps

module counter11 (
    clock,           // Input --> clock for counter
    up_downN,        // Input --> 1 = counts UP, 0 = counts DOWN
    count_enb,       // Input --> enables counting when HIGH
    load,            // Input --> loads counter when HIGH
    load_data,       // Input --> data to be loaded into counter
    reset,           // Input --> asynchronous reset to all 0's
    count            // Output --> output 11-bit count value
);

//-----
// define inputs/outputs
//-----

```

[illegible]

### Description of I/Os

signal	type	description
clock	I	System clock in
start	I	Initiates a multiply when HIGH
ina	I	Multiply input A
inb	I	Multiply input B
product	O	22-bit multiply output IIIIIIIIIII.FFFFFFFF

```

*****/
`timescale 1 ns / 10 ps

module multiply11x11 (clock, reset, start, ina, inb, done, product);

//-----
// define inputs/outputs
//-----

input  clock, reset, start;
input  [10:0] ina, inb;
output done;
output [21:0] product;

wire [21:0] adder_A, adder_B;

reg done;
reg [1:0] state;
reg [3:0] counter;
reg [10:0] latch_A, latch_B;
reg [21:0] product;

//-----
// define constants
//-----

`define NUM_CYCLES 4'b1011
`define IDLE_STATE 2'b00
`define LOAD_STATE 2'b01
`define MULT_STATE 2'b11

//-----
// assign statements
//-----

assign adder_A[21:11] = 0;
assign adder_A[10:0]  = (latch_B[10] == 0) ? 0 : latch_A;
assign adder_B = product << 1;

//-----
// synchronous state machines w/async reset
//-----

always @ (posedge clock or posedge reset)
begin
    if (reset == 1)
    begin
        state    <= `IDLE_STATE;
        counter  <= 0;
        product  <= 0;
        latch_A  <= 0;
        latch_B  <= 0;
        done     <= 0;
    end
    else

```

```

begin
  case(state)
    `IDLE_STATE :
      begin
        if (start == 1)
          begin
            state  <= `LOAD_STATE;
            counter <= `NUM_CYCLES;
            product <= 0;
            latch_A <= ina;
            latch_B <= inb;
            done    <= 0;
          end
        else
          begin
            state  <= `IDLE_STATE;
            counter <= 0;
            product <= product;
            latch_A <= 0;
            latch_B <= 0;
            done    <= done;
          end
        end
      end
    `LOAD_STATE :
      begin
        state  <= `MULT_STATE;
        counter <= counter - 1;
        product <= adder_A + adder_B;
        latch_A <= latch_A;
        latch_B <= latch_B << 1;
        done    <= 0;
      end
    `MULT_STATE :
      begin
        if (counter == 0)
          begin
            state  <= `IDLE_STATE;
            counter <= 0;
            product <= product;
            latch_A <= 0;
            latch_B <= 0;
            done    <= 1;
          end
        else
          begin
            state  <= `MULT_STATE;
            counter <= counter - 1;
            product <= adder_A + adder_B;
            latch_A <= latch_A;
            latch_B <= latch_B << 1;
            done    <= 0;
          end
        end
      end
    default :
      begin
        state  <= `IDLE_STATE;
        counter <= 0;
        product <= 0;
        latch_A <= 0;
        latch_B <= 0;
      end
  end
end

```

```

        done    <= 0;
    end

endcase

end

end

endmodule

/*****
*          Sigma-Delta Pulse-Width Modulator (SDPWM)          *
*                                                              *
*   File      : pngen11.v                                     *
*   Module(s) : pngen11                                       *
*   Engineer  : Richard S. Lewison                           *
*   Date      : December 28, 1996                             *
*              February 10, 1997                             *
*                                                              *
* ----- Description ----- *
This module is an 11-bit pseudo-random number generator. This PN generator uses
a linear-shift register with taps defined by the constant MASK. The width of the
PN number is 4 bits. From reset, the LSR is set to zero. Once reset is de-
activated, the LSR cycles through 2^N combinations. The LSR uses 0 as a
valid state and decodes it to a non-zero state in order to get 2^N valid states.
*****/
`timescale 1 ns / 10 ps

module pngen11 (
    clock, // Input - System clock for PN generator
    enable, // Input - Trigger to generate new 11-bit PN
    reset, // Input - Asynchronous reset of all registers
    pn      // Output - 11-bit pseudo-random number
);

//-----
// Define inputs/outputs
//-----

input  clock, enable, reset;
output [10:0] pn;

//-----
// Define Signals
//-----

wire bit0;
reg  [10:0] pn;

//-----
// Define Constants for Tap Points
//-----

`define MASK 11'b10010001001 // This is the LFSR tap mask
`define LAST 11'b10101010101 // This is the USER-DEFINED break before ZERO
`define FIRST ((`LAST < 1) & 11'b1111111111) | {10'b0000000000, (^(`MASK & `LAST))} //
After ZERO

//-----
// Assign statement(s)
//-----

assign bit0 = (^(pn & `MASK));

```





```

module pulse_inv (
    enable,          // In - enables pulse inverting
    pulse_count,     // In - pulse count
    max,             // In - maximum possible pulses (programmable)
    duty_cycle,      // Out - resultant PWM pulse duty cycle
    delay,           // Out - resultant PWM pulse delay
    polarity         // Out - resultant PWM pulse polarity
);

//-----
// Inputs & outputs
//-----
input  enable;
input  [10:0] pulse_count, max;
output [10:0] duty_cycle, delay;
output polarity;

wire [10:0] half_max;
reg  [10:0] delay, duty_cycle;  // pseudo-registers

//-----
// ASSIGN statements
//-----
assign half_max = (max >> 1);
assign polarity = (enable & (pulse_count > half_max)) ? 1 : 0 ;

//-----
// Determine POLARITY and PULSE_COUNT asynchronously
//-----
always @ (polarity or pulse_count)
begin

    if (polarity == 1'b0)
    begin
        delay      = max - pulse_count;
        duty_cycle = pulse_count;
    end
    else
    begin
        delay      = pulse_count;
        duty_cycle = max - pulse_count;
    end
end

endmodule

/*****
*          Sigma-Delta Pulse-Width Modulator (SDPWM)          *
*                                                              *
*   File       : TOP_digital.v                                *
*   Module(s)  : TOP_digital                                  *
*   Engineer   : Richard S. Lewison                           *
*   Date       : April 25, 1997                                *
*                                                              *
* ----- Description ----- *
* This module is the top-level digital module for the Delta Sigma Pulse Width *
* Modulator (DSPWM). *
*****/
`timescale 1 ns / 10 ps

module TOP_digital (
    clock,          // In - clock input from crystal oscillator
    reset,          // In - Asynchronous reset input (active high)

```

```

        dither_enb,      // In - Dither enable flag (0 = no dither)
        SW,              // In - DIP switch setting for max[10:7]
        DSM_out,         // In - output from Delta-Sigma modulator
        PWM_out          // Out - DSPWM output (PWM waveform)
    );

//-----
// Inputs & outputs
//-----

input [3:0] SW;
input clock, reset, dither_enb, DSM_out;
output PWM_out;

reg polarity_reg, PWM_out, reset_1, reset_2, reset_sync;
reg [10:0] pn_reg, pulse_cnt_reg;
wire clock, last_pulse, first_pulse, second_pulse, third_pulse;
wire start_PWM, delay_enb, duty_enb;
wire DSM_out, reset, dither_enb;
wire [3:0] SW;
wire [10:0] max_clock_cnt, max_pulse_cnt, clock_cnt, pulse_cnt, delay_final;
wire [10:0] delay, duty_cycle, pn, delay_cnt, duty_cnt, pc_load_data;
wire [11:0] delay_plus1;
wire [21:0] delay_rand;

//-----
// ASSIGN statements
//-----

assign max_clock_cnt = {SW,7'b1111110};           // maximum clock count (program-
mable)
assign max_pulse_cnt = {SW,7'b1111111};           // maximum pulse count (program-
mable)
assign last_pulse = (clock_cnt == max_clock_cnt); // LAST pulse
assign first_pulse = (clock_cnt == 0);             // FIRST control pulse
assign second_pulse = (clock_cnt == 1);            // SECOND control pulse
assign third_pulse = (clock_cnt == 2);            // THIRD control pulse
assign delay_plus1 = {1'b0,delay} + 12'b1;        // delay plus one
assign delay_final = (delay_plus1[11]) ? pn_reg : delay_rand[21:11]; // use straight pn if
carry
assign delay_enb = (delay_cnt != 0);               // enable pulse delay count-down
assign duty_enb = ((delay_cnt == 0)&(duty_cnt != 0)); // enable duty cycle count-down
assign start_PWM = (clock_cnt == 25);             // time to trigger a new PWM cycle
assign pc_load_data = {10'b0,DSM_out};            // data to be loaded into pulse
counter

//-----
// Instantiate other modules
//-----

counter11 clock_counter (
    .clock(clock),           // --- clock counter ---
    .up_downN(1'b1),        // Input --> clock for counter
    .count_enb(1'b1),       // Input --> 1 = counts UP, 0 = counts DOWN
    .load(last_pulse),      // Input --> enables counting when HIGH
    .load_data(11'b0),      // Input --> loads counter when HIGH
    .reset(reset_sync),     // Input --> data to be loaded into counter
    .count(clock_cnt),      // Input --> asynchronous reset to all 0's
    .output(11'b0)          // Output --> output 11-bit count value
);

counter11 pulse_counter (
    .clock(clock),           // --- pulse counter ---
    .up_downN(1'b1),        // Input --> clock for counter
    .count_enb(1'b1),       // Input --> 1 = counts UP, 0 = counts DOWN
    .load_data(11'b0),      // Input --> data to be loaded into counter
    .reset(reset_sync),     // Input --> asynchronous reset to all 0's
    .count(clock_cnt),      // Input --> asynchronous reset to all 0's
    .output(11'b0)          // Output --> output 11-bit count value
);

```

```

        .count_enb(DSM_out),          // Input --> enables counting when HIGH
        .load(first_pulse),           // Input --> loads counter when HIGH
        .load_data({10'b0,DSM_out}), // Input --> data to be loaded into counter
        .reset(reset_sync),           // Input --> asynchronous reset to all 0's
        .count(pulse_cnt)             // Output -> output 11-bit count value
    );

pulse_inv pulse_inv (                // --- pulse inversion logic ---
    .enable(dither_enb),              // Input --> enables dithering
    .pulse_count(pulse_cnt_reg),      // Input --> registered pulse count
    .max(max_pulse_cnt),              // Input --> max pulse count
    .duty_cycle(duty_cycle),          // Output -> PWM pulse duty cycle
    .delay(delay),                    // Output -> PWM pulse delay (max)
    .polarity(polarity)              // Output -> PWM pulse polarity
);

pngen11 pngen (                     // --- pseudorandom number generator ---
    .clock(clock),                   // Input - System clock for PN generator
    .enable(start_PWM),              // Input - Trigger to generate new 11-bit PN
    .reset(reset_sync),              // Input - Asynchronous reset of all registers
    .pn(pn)                          // Output - 11-bit pseudo-random number
);

multiply11x11 mult (                // --- pulse delay multiplier ---
    .clock(clock),                   // Input - System clock
    .reset(reset_sync),              // Input - Asynchronous reset
    .start(third_pulse),             // Input - Start multiply
    .ina(delay_plus1[10:0]),          // Input - operand #1 = delay plus one
    .inb(pn_reg),                    // Input - operand #2 = registered pn
    .done(),                          // Output - flag 1=done (unused here)
    .product(delay_rand)             // Output - 22-bit product
);

counter11 duty_timer (               // --- pulse duty cycle timer ---
    .clock(clock),                   // Input --> clock for counter
    .up_downN(1'b0),                 // Input --> 1 = counts UP, 0 = counts DOWN
    .count_enb(duty_enb),            // Input --> enables counting when HIGH
    .load(start_PWM),                 // Input --> loads counter when HIGH
    .load_data(duty_cycle),           // Input --> data to be loaded into counter
    .reset(reset_sync),              // Input --> asynchronous reset to all 0's
    .count(duty_cnt)                 // Output -> output 11-bit count value
);

counter11 delay_timer (              // --- pulse delay timer ---
    .clock(clock),                   // Input --> clock for counter
    .up_downN(1'b0),                 // Input --> 1 = counts UP, 0 = counts DOWN
    .count_enb(delay_enb),           // Input --> enables counting when HIGH
    .load(start_PWM),                 // Input --> loads counter when HIGH
    .load_data(delay_final),          // Input --> data to be loaded into counter
    .reset(reset_sync),              // Input --> asynchronous reset to all 0's
    .count(delay_cnt)                // Output -> output 11-bit count value
);

//-----
// Synchronous logic
//-----

always@(posedge clock or posedge reset_sync)
begin
    if (reset_sync == 1) begin
        polarity_reg <= 0;
        PWM_out       <= 0;
        pn_reg        <= 11'b0;
        pulse_cnt_reg <= 11'b0;
    end
end

```

```

end
else begin
    if (start_PWM) polarity_reg <= polarity;
    if (first_pulse) pulse_cnt_reg <= pulse_cnt;
    if (second_pulse) begin
        if (!dither_enb | (polarity^polarity_reg)) pn_reg <= 11'b0;
        else
            pn_reg <= pn;
        end
        PWM_out <= polarity_reg^duty_enb;
    end
end
end

//-----
// Synchronize Input "reset" to create system reset "reset_sync".
//-----

/* synopsys dc_script_begin
set_register_type -exact -flip_flop SYNCFF reset_1_reg
*/

always @ (posedge clock)
begin: sync_1
    if (reset == 1) reset_1 <= 1;          /* step 1 */
    else
        reset_1 <= 0;          /* step 1 */
end

always @ (posedge clock or posedge reset)
begin: sync_2
    if (reset == 1) reset_2 <= 1;          /* asynchronous reset */
    else
        reset_2 <= reset_1;          /* synchronize un-reset (step 2) */
end

always @ (posedge clock or posedge reset)
begin: sync_3
    if (reset == 1) reset_sync <= 1;          /* asynchronous reset */
    else
        reset_sync <= reset_2;          /* synchronize un-reset (step 3) */
end

endmodule

/*****
*
* File      : TOP_analog.v
* Module(s) : TOP_analog
* Engineer  : Richard S. Lewison
* Date      : February 6, 1997
*           : February 10, 1997
*
* ----- Description -----
* This module implements the top-level analog block of the DSPWM.
*****/
`timescale 1 ns / 10 ps

module TOP_analog(
    Vinp,          // InOut - Analog input signal (+)
    Vinn,          // InOut - Analog input signal (-)
    Vref,          // In - Analog ground (reference)
    clk1P,         // Phase 1 (+) switch capacitor clock
    clk1N,         // Phase 1 (-) switch capacitor clock
    clk2P,         // Phase 2 (+) switch capacitor clock
    clk2N,         // Phase 2 (-) switch capacitor clock
    reset,         // Input --> asynchronous reset
    DSM_out,       // Output --> modulator digital output
    AVDD,          // InOut - Analog VDD

```

```

        AGND                // InOut - Analog GND
    );

input clk1P,clk1N,clk2P,clk2N,Vinp,Vinn,Vref,reset;
output DSM_out;
inout AVDD, AGND;

reg DSM_out, up_downN;
wire clk1P,clk1N,clk2P,clk2N,Vinp,Vinn,Vref,reset;
real vin, vin_int, sum, feedback;
tri AVDD, AGND;

parameter top    = 1.5;          // -1 <= bottom <= top <= +1
parameter bottom = -1.5;         // -1 <= bottom <= top <= +1
parameter step   = 0.001;       // step size

//-----
// Simulation initialization
//-----
initial
begin
    $display("Starting Delta-Sigma modulator!");
    vin_int = bottom;
    up_downN = 1;
end

//-----
// Delta-Sigma Modulator
//-----
always@(posedge clk1P or posedge reset)
begin

    if (reset) begin
        DSM_out = 0;
        feedback = 0.0;
        vin_int = bottom;
        vin = -0.9999;
        up_downN = 1;
        sum = 0.0;
    end

    else begin
        //-----
        // Update Delta-Sigma Modulator
        //-----
        sum = sum + vin - feedback;
        if(sum >= 0.0) DSM_out = 1;
        else DSM_out = 0;
        if(DSM_out == 1) feedback = 1.0;
        else feedback = -1.0;

        // $display("Vin = %8.4f Up_DownN = %b Sum = %8.4f DSM_out = %b Feedback = %8.4f",
        //         vin,up_downN,sum,DSM_out,feedback);

        //-----
        // Update Stimulus
        //-----
        case(up_downN)
            1'b0: begin
                if(vin_int-step > bottom) vin_int = vin_int - step;
                else begin
                    vin_int = bottom;
                    up_downN = 1;
                end
            end
        endcase
    end
end

```

```

        end

        1'b1: begin
            if(vin_int+step < top) vin_int = vin_int + step;
            else begin
                vin_int = top;
                up_downN = 0;
            end
        end
    endcase
    if (vin_int>=1.0) vin = 1.0;
    else if (vin_int<=-1.0) vin = -1.0;
    else
        vin = vin_int;
    end
end
endmodule

```