Final Project - Waveform Generator

CEG3155[A] - Digital Systems II
Fall 2018

School of Electrical Engineering and Computer Science

University of Ottawa

Course Coordinator: Miodrag Bolic

Aron Arany-Takacs - 8733338 - aaran043@uottawa.ca

Jordan Benoit - 8346159 - jbeno064@uottawa.ca

Spencer Hayes-Laverdiere - 8614154 - shaye059@uottawa.ca

Experiment date: December 6th, 2018

Submission Date: December 10th, 2018

Contents

The two main strategies of waveform generation for most of the existence of the synthesizer have been the utilization of Oscillators or wavetables. Oscillators are typically standalone electrical components that can generate a wave function in terms of analogue voltage, or bits depending on the implementation. This waveform is generated upon use. Wavetables, on the other hand, use pre-synthesized values of a wave along certain data points (our synthesizer belongs in this category). Beyond the basic concept, more techniques exist to create complex waveforms from the basics provided by these two methods.

Two common examples of these techniques are additive and subtractive synthesis. Additive synthesis involves adding waveforms over one another, to produce complex overtones (sounds of different frequencies overlapping to form harmonics). Subtractive synthesis is the opposite, applying a series of filters and destructive interference to modify the shape of the waveform beyond a simple sine, square, saw or triangle wave.

For most of the development of early synths in the 70s and 80s, however, Frequency-Modulated (FM) synthesis was the most popular. FM synthesizers use two or more oscillators operating at differing frequencies and amplitudes to stack and combine waveforms that could more accurately simulate analogue sound than the basic methods. The waves generated could also be modified further by additive and subtractive synthesis, increasing the melodic depth and potential of the chip. Phase synthesis was also common in early synths, stretching and phase-shifting existing wavetables to produce varied sound.

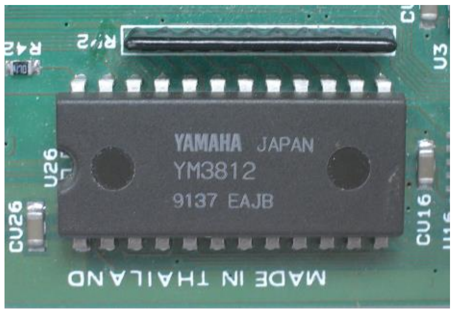


Fig. 1 - an FM YM3812 CPU

In the modern era, FM synthesis has largely been replaced by sample-based synthesis, mainly due to the exponential growth of computer memory space (to accommodate high-bitrate samples) and the more accurate and varied sound that could be produced by this technique. Sample-based synthesis uses digital audio samples, recorded from analogue sources or other high-fidelity methods, and then modifies it further using filters while keeping the original audio intact in memory. The potential of modern sample-based synthesizers is much deeper than older FM chips like the Yamaha YM3812 or the Casio CZ-series of copycat synths, although many prefer and enjoy the ‘retro’ or digital flavour of the older methods, with older classic keyboards becoming sought-after collector’s pieces.

Extended Techniques & Scope of the Project *(ii)*

In addition to the waveform generation methods outlined in the previous section, the modulation of an existing waveform in basic structure can be outlined by ADSR, or Attack, Decay, Sustain and Release. These attributes define the structure of the waveform, and are applied once the wave has already been generated. Attack refers to the speed at which the peak amplitude of the wave is reached, the decay refers to the slope that the wave degrades into after the peak, the sustain is the amplitude it stabilizes at, and the release is the slope of the fade after the note ends.

Numerous other effects can be added to the wave post-generation, including bitcrushing, low and hi-pass filters, echo, phase shifting, paulstretching and more. These effects change other qualities of the sound. Bitcrushing staggers the wave by drastically decreasing the sample rate, giving other waves similar qualities to a square wave in comparison. Low and hi-pass filters cull the waveform at the two extremes, decreasing the tinniness or the bass of the sound to achieve the effect similar to a telephone call, or hearing noise from another room. Many other effects like these exist and can be applied to waves not just by the synthesizer, but by third-party software and audio DAWs to enhance the recorded sound.

In this project, however, these effects are limited due to time and complexity restraints. Such complex filters require intricate equations and transformations defined by algorithms, all of which require time and effort to implement. To this end, this project requires simply implementing a wavetable-based synthesizer that outputs to an external DAC, using simple square, sine, triangle and saw waves without complex modulation, save for defining the period and sample rate.

Theoretical Concept of the Synthesizer *(iii)*

For our synthesizer, a basic structure was outlined in the requirements (*Fig. 2*). An FSM would take control inputs, and translate them into commands for a counter, memory module and data latch in order to properly sequence transmission into an external DAC. The FSM would have two main control bits ST and RSR (start and reset respectively) that would enable the waveform generation process and reset the current output to an idle state no matter the position of the circuit. In addition, the FSM would receive input from four other control inputs, namely the select inputs of the chosen waveform (ch_W), period (ch_P), sample rate (ch_SR) and the *full* flag supplied by the counter module.

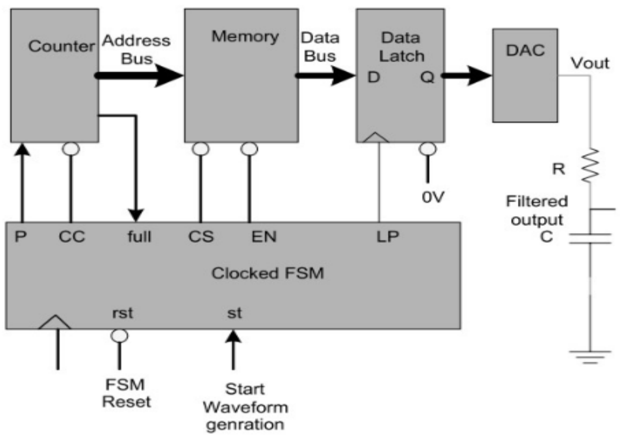


Fig. 2 - the Given Conceptual Outline of the Synthesizer

The counter, receiving the input of the chosen waveform, would map the 2-bit control input to the starting value of a memory address located in the memory module. It would then select locations to be transmitted to the latch on intervals in sequence with the sample rate and period of the waveform. This data would then be stored on the D-latch, then transmitted to the DAC for processing into an analogue signal, likely an audio jack or external speaker.

The primary functions of each module would be; the coordination of memory read flags by the **Counter**, the storage of wavetable data in the **Memory**, the implementation of the sample rate by the **Data Latch**, and the regulation of signal timing and coordination by the **Clocked FSM**.

Memory Module *(iv)*

The memory module of the synthesizer is implemented as a 3D array of 8-bit memory locations, each storing an unsigned integer representing the amplitude of the chosen waveform. The memory array is divided into 4 sets of 256 memory locations, with the first two bits of the 10-bit address referring to [0, 1, 2, 3], the sections corresponding to the stored waveforms (Sine, Square, Triangle and Saw respectively). Each of these locations has bytes 0...255 storing the amplitude data generated by a python function, taking input from the individual equations for each wave:

Waveform Section	Amplitude Data Sections
SINE - 00 [0]	[00000000] 0, 1, 2, 3...255 [11111111]
SQUARE - 01 [1]	[00000000] 0, 1, 2, 3...255 [11111111]
TRIANGLE - 10 [2]	[00000000] 0, 1, 2, 3...255 [11111111]
SAWTOOTH - 11 [3]	[00000000] 0, 1, 2, 3...255 [11111111]

Fig 3. Table Representation of Array Addresses

The Memory module transmits a location using a request variable, supplied by the counter module. On each clock rising edge, the memory reads an *addr_request* 10-bit signal from the counter. The module then maps it to the corresponding waveform section by isolating the two m.s.b.s, and then using the other 8 bits to map it to the corresponding array location. The data from this location is then sent out through a *d_out* variable to the Data Latch for processing. If the counter does not change the address requested, the module simply reroutes to itself for another clock cycle.

Counter Module *(v)*

The Counter module's purpose is to index the selected waveform memory location from a given address, and send a signal *full* to the FSM once the entire waveform has been processed. It does this by setting a counter to start at the value of the location of the first address in memory (provided by the signal *p*). The counter increments until the final address has been reached (*p+255*) and then resets back to the originally provided *p*, while also setting *full* to '1'. The counter module does not take direct input from the clock, ***and is instead*** incremented on the rising edge of the *cc* signal, calculated by the COUNT state of the FSM. The address of *p* is also assigned in the FSM, so the counter can index independently of the current waveform selected.

The FSM, in this case a Moore machine, is responsible for the timing and state transitions of the component. It receives all the direct inputs from the control bits and flags, and transmits the inputs and read/write flags to all of the other components. To achieve this, three main state categories were implemented:

- IDLE
    - *This state is the natural resting state of the circuit,* ***RST('1')*** *naturally triggers a reset to here from any state during a clock cycle, and the state will be stable until* ***ST*** *is triggered*
    - *While in* IDLE*, all enable flags (****CC, EN, LP****) are set to '0' to prevent data transfer in the system*
    - *Once* ***ST*** *is triggered, the FSM scans for* ***ch_WF*** *input, and will transition to a dedicated Waveform state based on the input*

- SINE/SQUARE/TRIANGLE/SAW
    - *Is the precursor to the actual transmission of the waveform data, essentially a primer state for the* COUNT *state. This state is returned to after every count-through of the data, and the assignments here make sure that a transfer of waveform, sample rate and period do not necessitate activation of* ***RST****.*
    - *The Counter data input* ***P*** *is assigned here, to the initial memory location of the current waveform state. CNTR and CNTR_MAX internal variables are also assigned, CTR is reset to 0 while CTR_MAX is calculated using the specified period* ***ch_P***
    - ***N_skip*** *is computed, dividing 256 by* ***ch_SR*** *to check how many addresses must be skipped for the current sample rate*
    - *After assignments, the state checks again for* ***ch_WF*** *input, and if it's consistent with the current assigned state, the FSM progresses to* COUNT*. If not, then the next state is assigned to the specified waveform, and the cycle repeats. This means that* ***ST*** *and* ***RST*** *do not have to be triggered to change waveforms.*

- COUNT
    - *This state is responsible for translating clock timing into an incremented flag delay in line with the specified period. It's a double-looping state, only reverting back to the waveform state when all the data point visits in the memory have been exhausted*
    - ***CS*** *and* ***CC*** *are set to '0' in the first stage. The counter then increments, and checks to see if the CNTR value has reached the value of CNTR_MAX, the arithmetic representation of the amount of clock cycles needed to reach 1/256th of the specified period. If false, it loops back to the beginning of the state. If true.* ***CC, CS*** *are set to '1', signifying a rising edge, and allowing for the transmission of data between the counter, memory and latch*
    - *The Data Latch Logic is also checked, as the* ***n_skip*** *is calculated with a counter incrementing for each change in data. If reached, the Data Latch is allowed to change the stored value*
    - *If* ***full*** *is '1', the counter checks for* ***ch_WF*** *and returns to the waveform state. If not, it repeats until all data points have been exhausted*

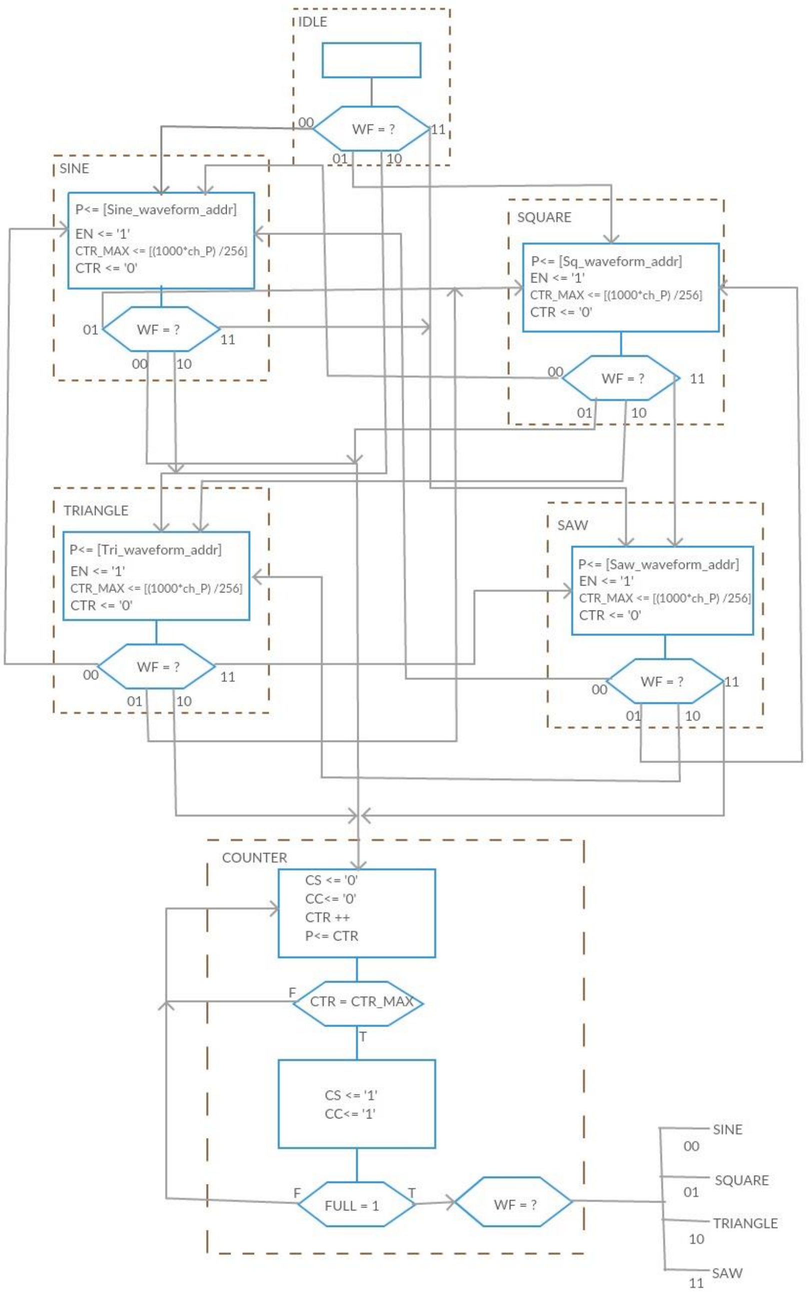


Fig. 4 - ASMD Chart of FSM_ctrl (IDLE state assignments not included to save page space, see. Main Section)

Theoretical Design of a DAC Using VHDL and Binary Logic *(vii)*

Though it wasn't necessary to design the DAC portion of the synthesizer, this was unclear at the start. Therefore, a design was drafted up for the implementation of such a component, although no VHDL code exists for the module. Due to the limitations of the language, VHDL cannot express a variable voltage as a variable, as it can only output binary logic as bits, not an integer range as a physical entity. Thinking around this limitation, as well as after some research, a theory was drawn up.

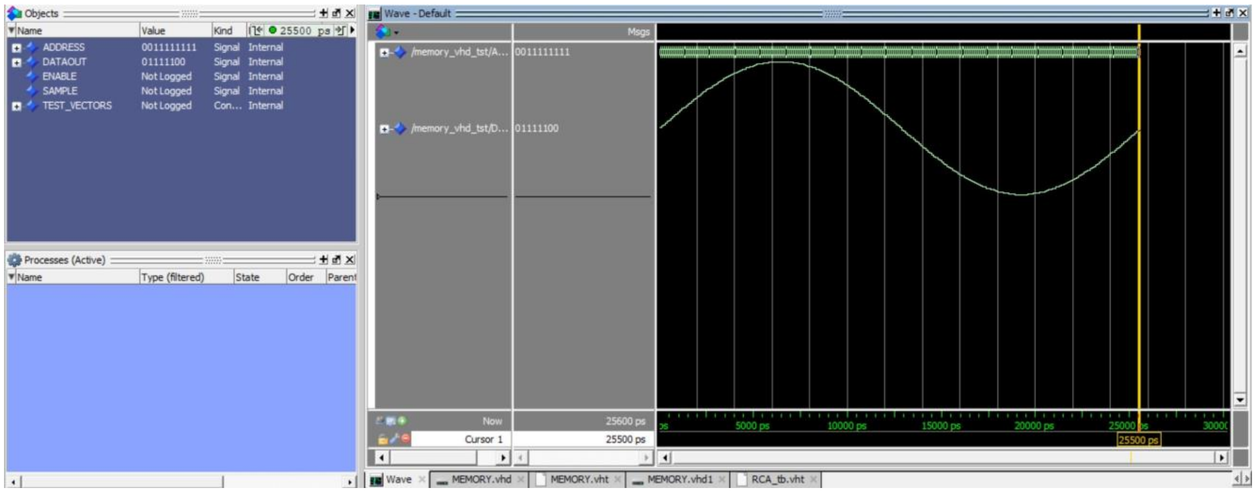
Though the output is limited (full on/off, 1 or 0), the output of a pin still has rise and fall time inherently, due to the properties of electrical circuits. Say the Power Supply Unit (PSU) could output +5V or 0V to a pin. If a component could rapidly oscillate between 0 and 1 at clock-frequency speed, the true output of the DAC could be +5V / 2 = +2.5V. This principal could be taken further, as a rapid oscillation in the sequence of [0, 0, 1] would yield a voltage of +5V / 3 = +1.666V.

Theoretically, this could mean that any voltage between 0V and +5V could be outputted by the circuit. Unfortunately, the algorithms for mapping bit values to the ratio of 1's and 0's needed would be relatively complex and far outside the scope of the project. This is especially difficult when it comes to ratios in the range of 27:32, 300:301, etc. in order to achieve the precise voltage control needed to properly and clearly output an analogue wave signal. In addition, the oscillations would take up a decent amount of clock periods, especially when coming up on >100 integer ratios, which would require at least 200 clock cycles to output a voltage level. Therefore, the ranges would be quite limited in feasibility, and perhaps more suited to a 4-bit amplitude than an 8-bit one.

Concluding Thoughts *(viii)*

The project went smoothly, and even more concepts were explored than outlined, such as the aforementioned DAC. Needless to say, improvements can be made in the code, as it was not perfect, and the three-person environment often caused troubles regarding combining logic into a top file, and certain portions had to be rewritten. The frequency and timing also depends heavily on the clock, which, while in an FPGA is consistent, can cause issues if utilized with other hardware, or pipelined in a larger circuit due to delay. Despite these constraints, the component functions in the specified state on the testbench, and can clearly output the required signals, as evidenced by the following section. The project was an excellent learning experience, proving a good introduction to a complex and relatively free-form circuit design.

Waveform Data and Testbench Output

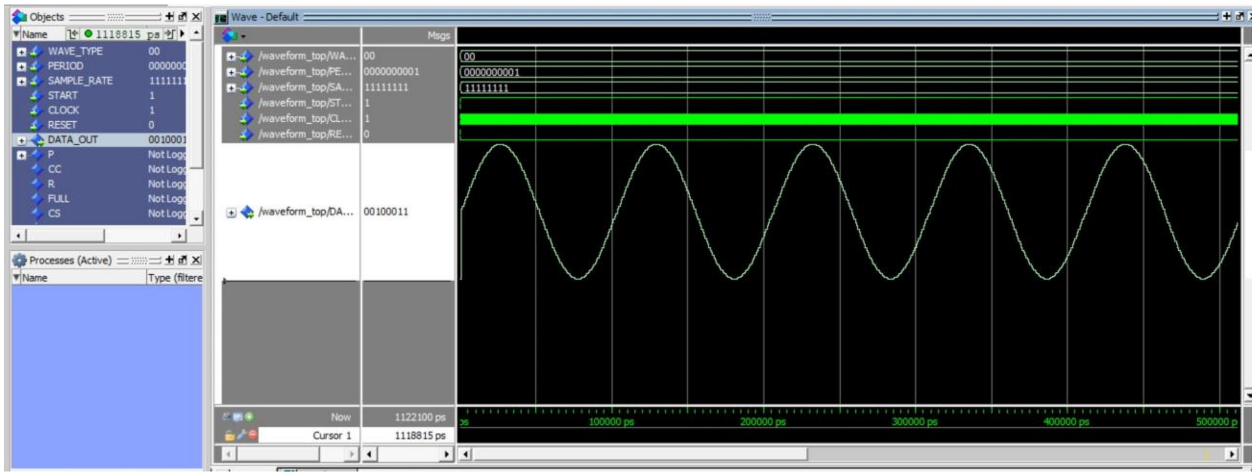


Name	Value	Kind	
ADDRESS	0011111111	Signal	Internal
DATAOUT	01111100	Signal	Internal
ENABLE	Not Logged	Signal	Internal
SAMPLE	Not Logged	Signal	Internal
TEST_VECTORS	Not Logged	Con…	Internal

	Msgs
/memory_vhd_tst/A…	0011111111
/memory_vhd_tst/D…	01111100

```
Now       25600 ps
Cursor 1  25500 ps
```

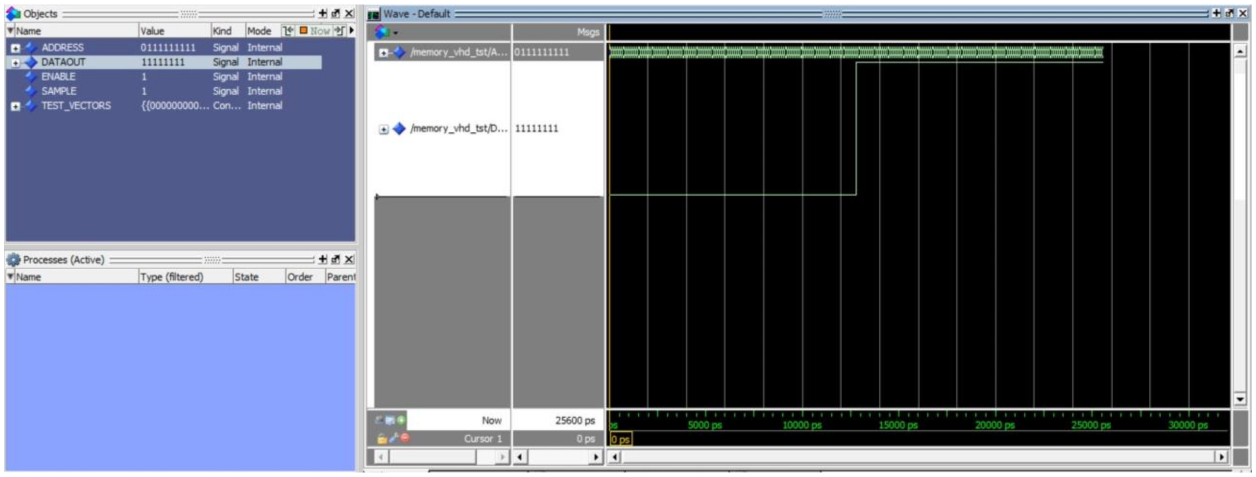
Sine Wave Testing & Output:



Name	
WAVE_TYPE	00
PERIOD	000000
SAMPLE_RATE	111111
START	1
CLOCK	1
RESET	0
DATA_OUT	0010001
P	Not Logg
CC	Not Logg
R	Not Logg
FULL	Not Logg
CS	Not Logg

	Msgs
/waveform_top/WA…	00
/waveform_top/PE…	0000000001
/waveform_top/SA…	11111111
/waveform_top/ST…	1
/waveform_top/CL…	1
/waveform_top/RE…	0
/waveform_top/DA…	00100011

```
Now       1122100 ps
Cursor 1  1118815 ps
```

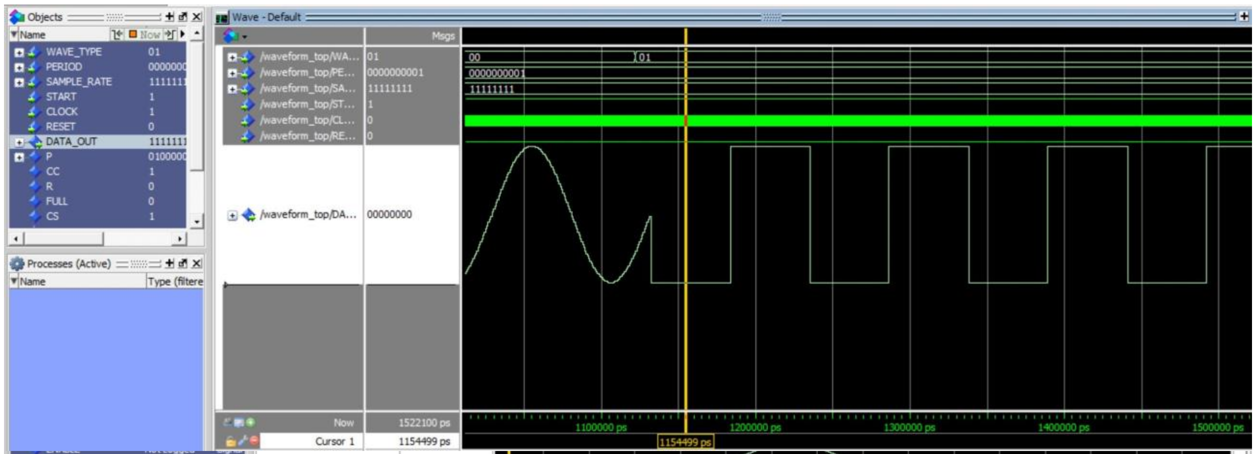


```
Objects
Name            Value            Kind     Mode
ADDRESS         0111111111       Signal   Internal
DATAOUT         11111111         Signal   Internal
ENABLE          1                Signal   Internal
SAMPLE          1                Signal   Internal
TEST_VECTORS    {{000000000...   Con...   Internal

Processes (Active)
Name   Type (filtered)   State   Order   Parent
```

```
Wave - Default
                          Msgs
/memory_vhd_tst/A...      0111111111
/memory_vhd_tst/D...      11111111

Now        25600 ps
Cursor 1   0 ps
```

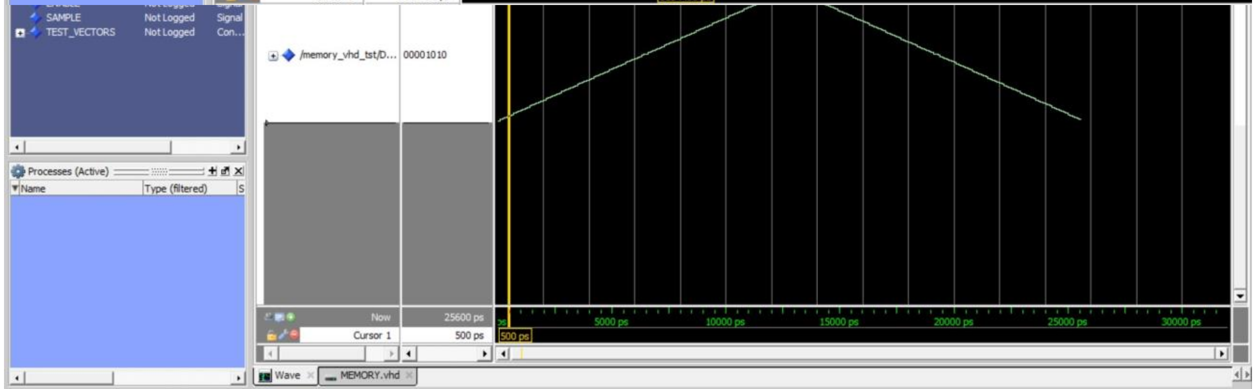
Square Wave Testing & Output:

Triangle Wave Testing & Output:



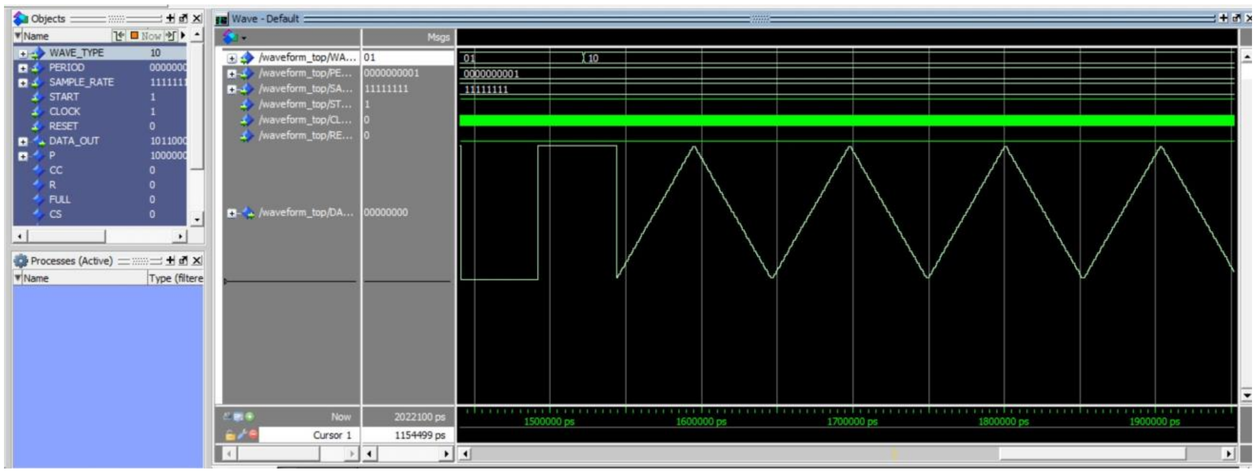
```
Objects
Name            
WAVE_TYPE       01
PERIOD          0000000
SAMPLE_RATE     1111111
START           1
CLOCK           1
RESET           0
DATA_OUT        1111111
P               0100000
CC              1
R               0
FULL            0
CS              1

Processes (Active)
Name   Type (filtere
```

```
Wave - Default
                          Msgs
/waveform_top/WA...       01
/waveform_top/PE...       0000000001
/waveform_top/SA...       11111111
/waveform_top/ST...       1
/waveform_top/CL...       0
/waveform_top/RE...       0
/waveform_top/DA...       00000000

Now        1522100 ps
Cursor 1   1154499 ps
```



```
SAMPLE          Not Logged   Signal
TEST_VECTORS    Not Logged   Con...

Processes (Active)
Name   Type (filtered)   S
```

```
/memory_vhd_tst/D...      00001010

Now        25600 ps
Cursor 1   500 ps
```

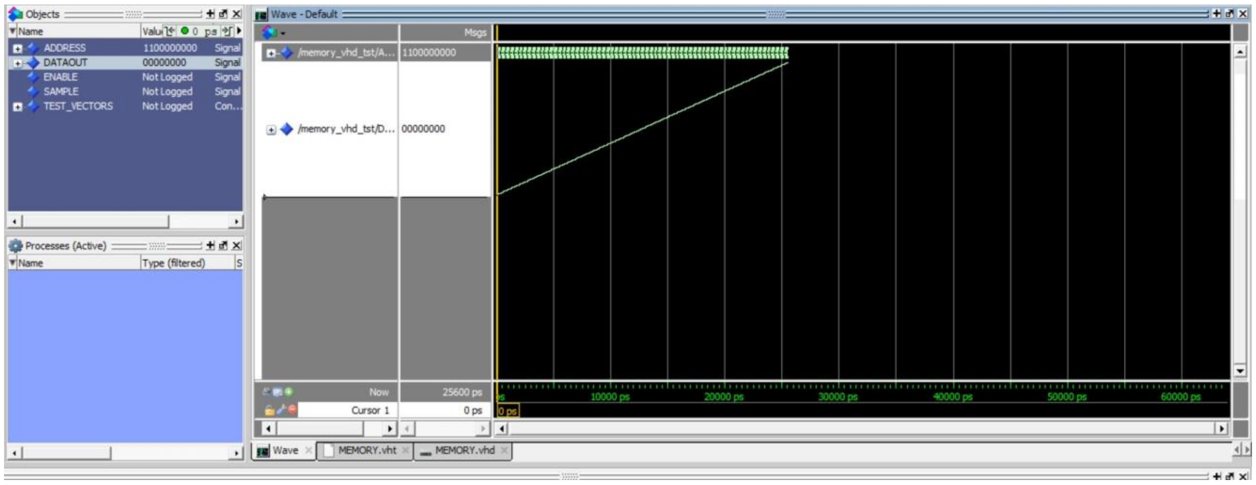


Name	Value
WAVE_TYPE	10
PERIOD	0000000
SAMPLE_RATE	1111111
START	1
CLOCK	1
RESET	0
DATA_OUT	1011000
P	1000000
CC	0
R	0
FULL	0
CS	0

Signal	Msgs
/waveform_top/WA...	01
/waveform_top/PE...	0000000001
/waveform_top/SA...	11111111
/waveform_top/ST...	1
/waveform_top/CL...	0
/waveform_top/RE...	0
/waveform_top/DA...	00000000

Now 2022100 ps
Cursor 1 1154499 ps

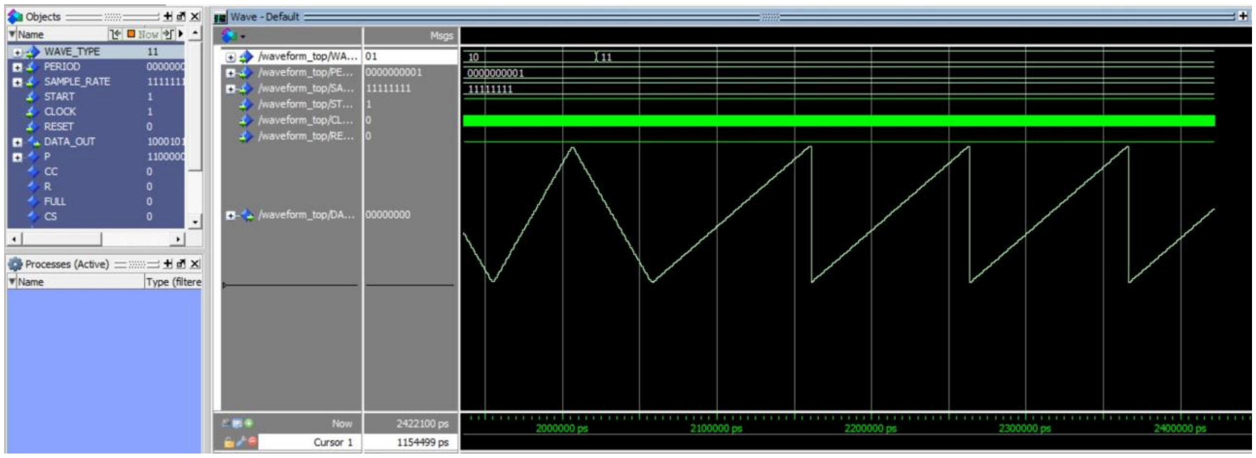
Sawtooth Testing & Output:



Name	Value	Kind
ADDRESS	1100000000	Signal
DATAOUT	00000000	Signal
ENABLE	Not Logged	Signal
SAMPLE	Not Logged	Signal
TEST_VECTORS	Not Logged	Con...

Signal	Msgs
/memory_vhd_tst/A...	1100000000
/memory_vhd_tst/D...	00000000

Now 25600 ps
Cursor 1 0 ps



Name	Value
WAVE_TYPE	11
PERIOD	0000000
SAMPLE_RATE	1111111
START	1
CLOCK	1
RESET	0
DATA_OUT	1000101
P	1100000
CC	0
R	0
FULL	0
CS	0

Signal	Msgs
/waveform_top/WA...	01
/waveform_top/PE...	0000000001
/waveform_top/SA...	11111111
/waveform_top/ST...	1
/waveform_top/CL...	0
/waveform_top/RE...	0
/waveform_top/DA...	00000000

Now 2422100 ps
Cursor 1 1154499 ps