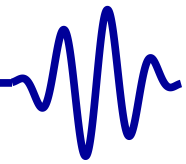
# 3. Tone Generation

Gisselquist
Technology, LLC

Daniel E. Gisselquist, Ph.D.

# Lesson Overview





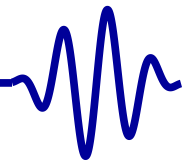
Objective: Building a basic memory mapped peripheral

- Build a register controlled bus slave
  - With multiple simple registers per slave
- Output an audio sine wave

Hardware:

- To build this project in hardware, you will need . . .
- A 1-bit audio amplifier
- An FPGA board with serial port control
  - We'll build off of the AutoFPGA demo from last time

# Work in Progress

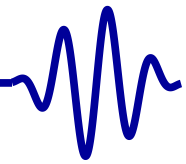




*This lesson is currently a work in progress.*


It will remain so until . . .

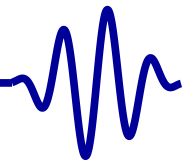
- I've added illustrations, and example course material
- I've built the design myself





# Tone Generator

# Audio Pipeline





Sound processing typically takes place at 200kHz or less

- Most FPGAs run at 20MHz or more
  - I typically run at 100Mhz
- We'll need to slow our logic down to process sound

Sound timing is often determined by a sample clock

- This can come from an external source, asynchronous to our system clock
- We'll be at the beginning of the sound processing chain, so
  - We can generate our own sample clock
  - We'll use 48kHz

The rule: Logic *only* steps forward once every 1/48kHz

# Clock Enables



**The rule**: Nothing moves forward except on a clock enable

- Let's let `audio_ce` be our audio clock enable signal

```
always @(posedge i_clk)
if (i_reset)
        // Clear pipeline
else if (audio_ce)
        // Pipeline data processing
// else
//      *NOTHING*
```

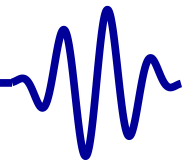
# Clock Enables



**The rule**: Nothing moves forward except on a clock enable

- You may eventually come across operations that take more work than can be done in one sample clock
  - You can then violate this rule
    - Examples: this FFT, or some filters
  - *Do so with care:* your core will then become dependent on the clock periods between sample clocks
  - This has consequences when it comes to reuse
- Our work today will (mostly) follow this rule

# Enable Generation



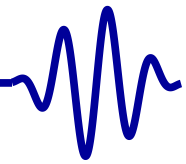


We'll need to generate a clock enable

- Much like we did for the 1PPS in the beginner's tutorial
- We'll use a fractional clock divider for accuracy

```
parameter real SAMPLE_RATE_HZ = 48.0e3; // 48 kHz
parameter [31:0] CLOCK_STEP =
        SAMPLE_RATE_HZ * 4.0 * (1<<30)
                * 1.0 / CLOCK_RATE_HZ;

reg [31:0]      srate_counter;

always @(posedge i_clk)
    { audio_ce, srate_counter }
        <= srate_counter + CLOCK_STEP;
```

# Enable Generation



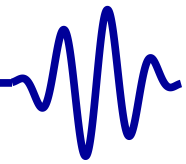


In case this looks unfamiliar

```
parameter real SAMPLE_RATE_HZ = 48.0e3; // 48 kHz
parameter [31:0] CLOCK_STEP =
        SAMPLE_RATE_HZ * 4.0 * (1<<30)
                * 1.0 / CLOCK_RATE_HZ;
```

- We want to calculate $2^{32}\frac{\texttt{SAMPLE\_RATE\_HZ}}{\texttt{CLOCK\_RATE\_HZ}}$
- By itself, 1<<32 will overflow any 32-bit integer
- 1<<31 is the maximum unsigned negative integer
  - Not what we want
- Multiplying by 4.0 converts 1<<30 to a real number
  - Specifically, it converts it to $2^{32}$

# Enable Generation

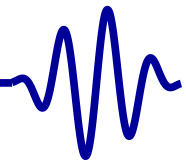




We'll need to generate a clock enable

- Much like we did for the 1PPS in the beginner's tutorial
- We'll use a fractional clock divider for accuracy
  - 32'bits will achieve 25mHz precision (that's *milli*-Hz)
  - Con: Phase noise
    - Clocks per enable will appear to jump between $\left\lfloor \frac{f_{\text{SYSCLK}}}{f_{\text{SAMPLE}}} \right\rfloor$ and $\left\lfloor \frac{f_{\text{SYSCLK}}}{f_{\text{SAMPLE}}} \right\rfloor + 1$ clocks
- Integer clock dividers may work as well
  - Con: Fewer potential frequency choices
    - Just how many clocks per enable would generate a 48kHz enable from a 100MHz system clock?
    - How about a 50MHz system clock? 25MHz?

# Tone generation





Imagine a big sinewave lookup table

- $2^N$ entries
- Let's assume for now that $N \rightarrow \infty$
  i.e. $N$ is *really* big
- Where $2^N$ entries captures one wavelength in the table

If we knew the phase of our sinewave

- We could look up the sine from the table

```
always @(posedge i_clk)
if (audio_ce)
        sin <= sintable[phase];
```

We could then adjust our phase to generate a tone

# Phase



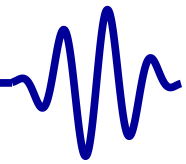
From one sample to the next, the phase would step forwards

```
always @(posedge i_clk)
if (audio_ce)
        phase <= phase + r_frequency_step;
```

- `phase` will automatically wrap at the end of one wavelength
- Checking for rollover at 360 degrees or $2\pi$ radians is no longer required

How should we set `r_frequency_step`?

# Frequency Step



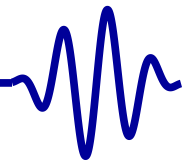


If `r_frequency_step` $= \frac{1}{2}2^N$

- Two steps will span the whole table before wrapping
- Can create outputs 1, -1, 1, -1, 1, -1


- This will generate a tone at the $1/2$ our sample rate
  This is also called the Nyquist frequency

# Frequency Step





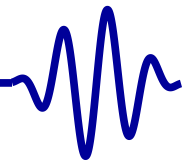
If `r_frequency_step` $= \frac{1}{2}2^N$

- Two steps will span the whole table before wrapping
- Can create outputs 1, -1, 1, -1, 1, -1


(or possibly 0,0,0,0 – see a DSP text)

- This will generate a tone at the $1/2$ our sample rate
  This is also called the Nyquist frequency

# Frequency Step

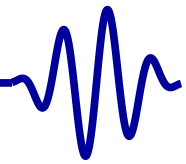




If `r_frequency_step` $= \frac{1}{4}2^N$

- Four steps will span the whole table before wrapping
- Each step will advance a quarter of the way through the table
- Will create outputs 0, 1, 0, -1, 0, 1, 0, -1, 0, 1, 0, -1
- This will generate a tone at the $1/4$ our sample rate

# Frequency Step



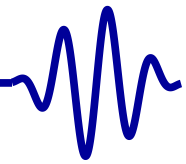


If `r_frequency_step` $= \frac{1}{8}2^N$

- Eight steps will span the whole table before wrapping
- Each step will advance one eigth of the way through the table
- Will create outputs 0, $\frac{\sqrt{2}}{2}$, 1, $\frac{\sqrt{2}}{2}$, 0, -$\frac{\sqrt{2}}{2}$, -1, -$\frac{\sqrt{2}}{2}$, 0, ...
- This will generate a tone at the $1/8$ our sample rate


Seeing a pattern?

# Frequency Step



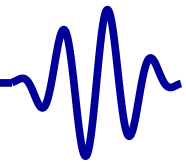


If `r_frequency_step` $= k2^N$, for $0 \leqslant k < \frac{1}{2}$

- We'll generate a tone at $k$ times our sample rate
  - If $f_s = 48 \times 10^3$
  - The generated tone will be at $kf_s$

But how many entries should be in our table?

# Table Size





What happens when the table size is smaller than $2^N$?

- There would be more phase bits than table entries

# Table Size



Suppose the table size is 2

- We'd then approximate the sine wave by a square wave

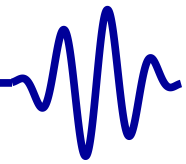

At what cost?

- We could skip the table lookup logic
- Just use the top bit of our phase

Whether this is "good enough" is application dependent

- I've used 1-bit tones for PLL inputs
- Old fashioned video games used to use 1-bit audio

# Table Size

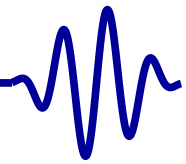




Suppose the table size is 4


This is better

# Table Size

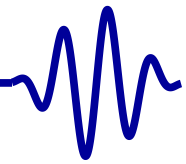




Suppose the table size is 8


This is even better yet

# Table Size



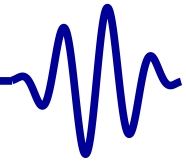


Suppose the table size is 16


Cost?

- 1LUT/bit, nearly free on iCE40s

# Table Size



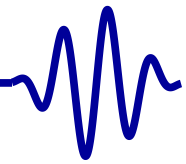


Suppose the table size is 64


Cost?

- 1LUT/bit, nearly free on Xilinx chips

# Table Size

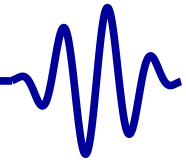




Suppose the table size is 256


Cost?

- 1 Slice/bit (4 LUTs), still quite cheap on Xilinx chips





The success of a given table size can be quantified

- Maximum error in time

$$\text{Max Err} = \max_{0 \leqslant t < 1} \sin(2\pi t) - \texttt{TBL}\left\lfloor 2^N t \right\rfloor$$

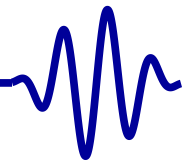
- Maximum spur energy in frequency
  - Measure the Fourier Series of one wavelength of the table

$$F_s(n) = \int_0^1 \texttt{TBL}\left\lfloor 2^N t \right\rfloor e^{-j2\pi n t} dt$$

  - Look for the largest distortion

$$\text{Max Spur Energy} = \max_n \left| \frac{F_s(n)}{F_s(1)} \right|^2$$

# Sinewave Generation

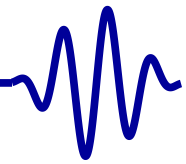




Several techniques for generating sine waves in hardware

1. Just use the most significant phase bit
   - Cheap and easy
   - Works nicely for PLLs
   - Can be used to generate FM signals near 100MHz
   - It might be good enough for your application
     - Old-fashioned arcade games used something similar
   - Not good enough for quality audio

# Sinewave Generation

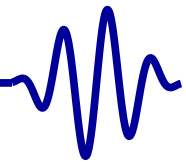




Several techniques for generating sine waves in hardware

1. Just use the most significant phase bit
2. Table lookup
   - Fairly cheap for $2^N < 512$.
     - Still fairly low logic
     - Larger sizes will use (precious?) block RAM resources
   - Quality might be "good enough"
   - Only generates a single amplitude
   - A 4096 point table can achieve $-70$dB maximum spur energy

# Sinewave Generation

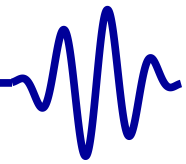




Several techniques for generating sine waves in hardware

1. Just use the most significant phase bit
2. Table lookup
3. CORDIC
   - Very well known approach
   - Doesn't use any multiplies
   - Can be made arbitrarily good
   - Logic usage becomes *very* expensive
     - For high precision
     - Especially when pipelined
       Sequential implementations can be much lighter

# Sinewave Generation

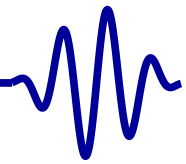




Several techniques for generating sine waves in hardware

1. Just use the most significant phase bit
2. Table lookup
3. CORDIC
   - Produces both sine and cosine outputs
   - Adjustable amplitude output
     - Result multiplies incoming value by sine/cosine
     - Algorithm introduces an additional scale
     - The internal scale factor may need compensation

# Sinewave Generation

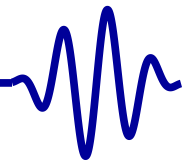




Several techniques for generating sine waves in hardware

1. Just use the most significant phase bit
2. Table lookup
3. CORDIC
4. Table + linear interpolation
   - Requires a multiply
   - Can greatly improve upon raw table lookups
   - Table can be built to ...
     - Acheive lowest sidelobe performance, or
     - Achieve minimum maximum error in time
   - A 64 point table can achieve $-70$dB maximum spur energy

# Sinewave Generation

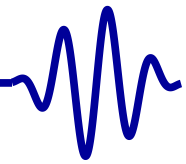




Several techniques for generating sine waves in hardware

1. Just use the most significant phase bit
2. Table lookup
3. CORDIC
4. Table + linear interpolation
5. Table + quadratic interpolation
   - Requires a two multiplies and several steps
   - 16 point table $\Rightarrow -70$dB maximum spur energy
   - 64 point table $\Rightarrow -180$dB maximum spur energy

# Sinewave Generation

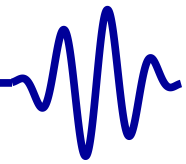




Several techniques for generating sine waves in hardware

1. Just use the most significant phase bit
2. Table lookup
3. CORDIC
4. Table + linear interpolation
5. Table + quadratic interpolation
6. Higher order approximations are possible
   - But are they really necessary?

# Sinewave Generation





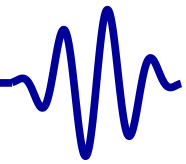
We'll use the basic table lookup method

- Building a lookup table is covered in the beginner's tutorial
  - You are welcome to build your own
  - My own example of sintable.v is also available
    This coregen can make designs for arbitrary bit-widths

```
always @(posedge i_clk)
if (audio_ce)
        sin <= sin_table[phase[31:20]];
```

Link: More advanced sinewave generators

# Magnitude Adjustment



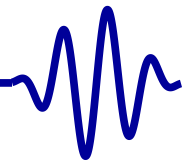


We'd like to be able to control how loud our tone is

- Multiply the table output by a scale factor

```
reg signed [15:0] sin, sample;
reg signed [16:0] r_scale;
reg signed [32:0] scaled;

always @(posedge i_clk)
if (audio_ce)
        sin <= sintable[phase[31:20]];

always @(posedge i_clk)
if (audio_ce)
begin
        scaled <= sin * r_scale;
        sample <= scaled[32:16];
end
```

# Bitwidth





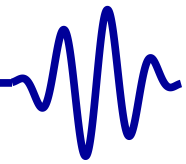
Bit widths must be carefully tracked in fixed bit math

- Adding two numbers increases the bit width by one over the largest incoming bit width
- Multiplying two numbers creates a result having the width of the sum of the two numbers
  - Beware of multiplying signed with unsigned numbers
  - To make `r_scale` signed, we'll need to keep the MSB clear

In our case

- We'll let our table have 16-bits, `r_scale` with 17-bits
- `scaled` will then have 33-bits
- The `sample` result will then have 16-bits again
- `r_scale` will scale the result from full scale to zero

# DSP Rules



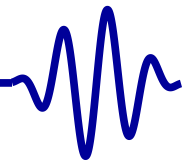


Many FPGAs have hard multiply accelerators

- These are often called *DSP*s
- Without hard DSP blocks
  - The synthesizer will try to pack all the multiplication logic in one clock cycle
  - This will consume a lot of logic
  - This may likely keep you from meeting timing as well
- If your FPGA doesn't have any DSPs, you may need to build an alternative
  - See here for a low-logic example
- As an engineer, you will need to manage DSP usage

Let's look at some rules to guarantee DSP allocation

# DSP Rules





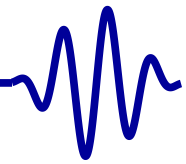
Rules for DSP usage

1. *Always* mark both operands as either signed or unsigned

```
reg signed [15:0]       sin;
reg signed [16:0]       r_scale;
```

   - Most math operations produce the same result independent of any operand sign: add, subtract, and, or, xor, etc
   - Multiplication is a key exception to this rule
   - Bit concatenations create unsigned values

# DSP Rules





Rules for DSP usage

1. *Always* mark both operands as either signed or unsigned
2. Like memories, keep the logic blocks sparse
   - Follow this form

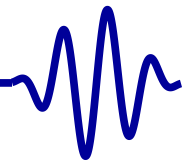
```
always @(posedge i_clk)
if (CE) // Put nothing else in this
        A <= X * Y; // logic block
```

   - Some hardware architectures allow an accumulate as well

```
always @(posedge i_clk)
if (CE)
        A <= X * Y + C;
```

   - Know what your hardware supports

# DSP Rules



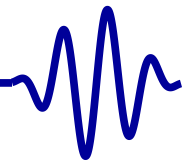


Rules for DSP usage

1. *Always* mark both operands as either signed or unsigned
2. Like memories, keep the logic blocks sparse
3. Resets should be kept external to the multiply
   - If you need a reset, reset the result on the next clock

```
initial { result, r_clear } = 1;
always @(posedge i_clk)
if (i_reset)
        { result, r_clear } = 1;
else if (r_clear)
        { result, r_clear } = 0;
else
        result <= multiply_output;
```

# Lag





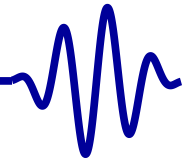
In many DSP applications, lag must only be bounded

- $\mu$s processing delays are irrelevant
- At $20$ ms, delays start to become relevant

My point:

> Taking a couple of clocks to clear the pipeline won't hurt

# Output





Our chosen hardware requires a one-bit audio output

- We need to drive this bit fom our hardware
- You may wish to use some of your registers to control power and amplifiers

Ex. The Pmod AMP2 has two amplifier controls to adjust: `o_shutdown_n` and `o_gain`

Four possibilities we'll consider to drive one audio bit

- 1'bit, using the MSB of our sample
- Pulse width modulation (PWM)
- Pulse density modulation (PDM)
- Delta Sigma modulation

# 1'bit



Easiest output solution: just produce top bit of every sample

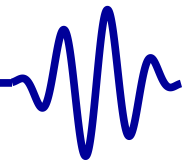
```
always @(*)
        o_audio = o_sample[15];
```

It works, but ...

- No way to adjust volume
- Not a pleasant sound
  - Equivalent to using a 1-bit sign table

We can do a lot better

# PWM

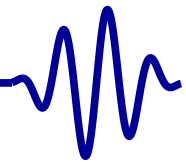




## Pulse Width Modulation (PWM)

- We could set `o_audio` on the first $\left(\texttt{o\_sample} + \frac{1}{2}\right) M$ of every $M$ clock periods
- Need to adjust the range of `o_sample`
  - Needs to be unsigned
  - Instead of a range from $-32,768, \ldots, 32,767$
  - Flip the MSB, to get a range from $0, \ldots, 65,535$

```
always @(posedge i_clk)
        pwm_counter <= pwm_counter + 1;

always @(posedge i_clk)
    o_audio <= (pwm_counter >=
        { !o_sample[15], o_sample[14:0] });
```



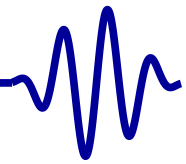


## Pulse Density Modulation (PDM)

- We could scramble which bits are set
- Reverse the order of bits in the `pwm_counter`
- Pushes the distortion to higher frequencies

```
genvar k;
generate for(k=0; k<PWM_BITS; k=k+1)
begin
    brev_counter[k] = pwm_counter[PWM_BITS-1-k];
end endgenerate

always @(posedge i_clk)
    o_audio <= (brev_counter >=
        ({ !o_sample[15], o_sample[14:0] });
```

# Delta-Sigma

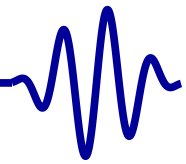




Delta Sigma modulation

- A formalized approach to PDM
- Can get a lot more complicated
- There's a lot of math behind this
  - Delta-Sigmal modulator is a control loop w/ feedback
  - First, second, and third order loops are possible

# Delta-Sigma



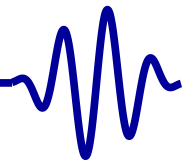


## Delta Sigma modulation

- A simple, first-order, delta-sigma modulator

```
reg      [15:0]  ds_counter;

always @(posedge i_clk)
    { o_audio, ds_counter }
        <= ({ !o_sample[15], o_sample[14:0] }
                + ds_counter;
```

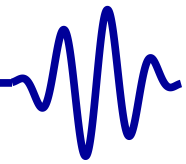
- We want high frequency, so we dropped the clock enable
- The carry from the addition is our output
- It's also naturally removed from the sum
  - This is the *delta* in delta-sigma modulation
  - The sum forms the *sigma*





# Debugging

# Debug visually



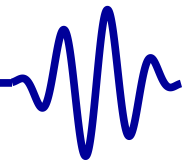


DSP is most easily debugged with pictures and graphs

- Step one: write the data to an external file

```
FILE *fp = fopen("dumpfile.16t", "w");
// ...
while (/* ... */) {
    // ...
    tick();
    if (m_core->o_audio_ce) {
        fwrite(&m_core->o_sample,
            sizeof(short), 1, fp);
        // May need to fflush this if fp ...
        // might be unexpectedly closed
        fflush(fp); // ... your call
    }
} // ...
fclose(fp);
```

# Octave script





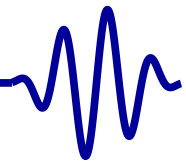
DSP is most easily debugged with pictures and graphs

- Matlab or Octave can really help here

You can then create an m-file script

```
fid = fopen('dumpfile.16t','r');
data= fread(fid, inf, 'int16t');
fclose(fid);

plot(data);
```

# Debug clock gating



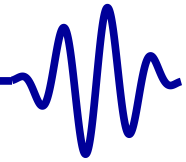


To gate on the clock enables or not?

```
if (m_core->o_audio_ce) // <-- gate here?
    fwrite(&m_core->o_sample,
        sizeof(short), 1, fp);
```

- Pro: Minimizes the data that needs to be examined
  - Useful if signals violate the clock enable rule(s)
  - Also if the clock enables aren't consistent
- Con: Might hide important events w/in your design
- Con: Relating multiple signals can be a challenge
  - Challenging example: resamplers

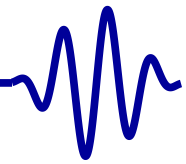
Whether or not you gate your output on `audio_ce` is a design decision





# AutoFPGA

# Bus connection

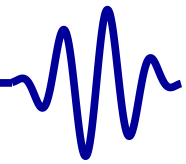




Let's connect this design to our bus

- We'll give it four registers
  1. Frequency step
  2. Scale factor
  3. (Reserved for your curiosity)
  4. (Reserved)

# DOUBLE slave





We'll use the AutoFPGA `SLAVE.TYPE=DOUBLE`

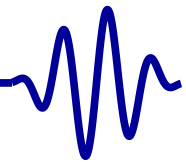
- Like `SINGLE`, `DOUBLE` uses a simplified bus interface
- `DOUBLE` slaves can have multiple registers
- Not allowed to stall the bus

```
always @(*)
        o_wb_stall = 1'b0;
```

- Require a single clock to generate any responses

```
initial o_wb_ack = 1'b0;
always @(posedge i_clk)
        o_wb_ack <= !i_reset && i_wb_stb;
```

# Bus writes





The rest of the bus logic is straight forward

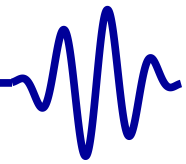
- We'll need to set our control registers

```
always @(posedge i_clk)
if (i_wb_stb && i_wb_we)
case(i_wb_addr)
2'b00: r_frequency_step <= i_wb_data;
2'b01: r_scale <= { 1'b0, i_wb_data[15:0] };
2'b10: begin end // Your option
2'b11: begin end // Your option
endcase
```

We can ignore the `i_wb_sel` lines

- Writing bytes or halfwords will produce *undefined* behavior
- This is not uncommon in digital design

# Bus reads



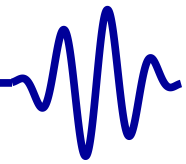


The rest of the bus logic is straight forward

- Also want to be able to read our control registers

```
always @(posedge i_clk)
case(i_wb_addr)
2'b00: o_wb_data <= r_frequency_step;
2'b01: o_wb_data <= { 16'h0, r_scale };
2'b10: o_wb_data <= 0; // Your option
2'b11: o_wb_data <= 0; // Your option
endcase
```

# AutoFPGA config





We'll also need an AutoFPGA configuration file

```
@PREFIX=tonegen
@NADDR=4                    Number of slave addresses
@SLAVE.BUS=wb               Connect to bus named wb
@SLAVE.TYPE=DOUBLE
@MAIN.PORTLIST=             Define a design port
```

```
        o_audio_ce, o_sample, o_audio
```

```
@MAIN.IODECL=   Declare our outputs
```

```
        output wire         o_audio_ce;
        output wire [15:0]  o_sample;
        output wire         o_audio;
```

# AutoFPGA Config

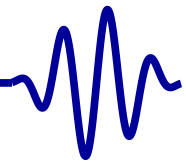


Our main level logic is inserted using the `@MAIN.INSERT` tag

`@MAIN.INSERT=` *Will be copied into main.v*

```
    @$(PREFIX)
    @$(PREFIX)i(i_clk, i_reset,
        // Connect to the bus
        @$(SLAVE.PORTLIST),
        // Connect our outgoing signals
        o_audio_ce, o_sample, o_audio);
```

# Register Definitions





We'll also tell our debug port about our registers

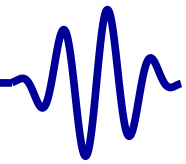
```
@REGS.N=2                                  Define two registers
@REGS.0=0 R_FREQUENCY FREQUENCY            Reg #1
@REGS.1=1 R_AMPLITUDE AMPLITUDE            Reg #2
```

The format of the `@REGS` tag:

- First field, the word offset of the register
- Then a C++ name for the register
- Finally, a user name (wbregs) for the register
- Additional names, if present, are additional usernames (aliases) for the same register

You can add more definitions if you choose to define more registers

# Makefile



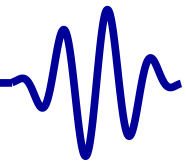


If your flow uses a Makefile when processing rtl/

- You can add `tonegen.v` to the list of dependencies used by this core

```
@RTL.MAKE.FILES=tonegen.v sintable.v   RTL files used
```

# Simulation tick





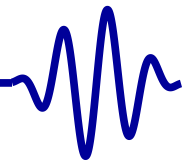
We'll want to add our debugging logic

- This will be called on every tick of the clock `clk`

```
@SIM.CLOCK=clk
@SIM.TICK=
```

```
    if (m_core->o_audio_ce) {
        fwrite(&m_core->o_sample,
            sizeof(short), 1, fp);
        fflush(fp); // ... your call
    }
```

# Top level I/Os

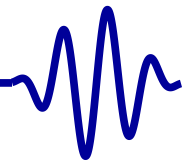




What about the top level I/O's?

- `o_audio_ce` is really an internal signal
- As is `o_sample`
- Without telling AutoFPGA otherwise
  - `@MAIN.PORTLIST` is used at the toplevel
  - As is `@MAIN.IODECL`
  - These will not just be outputs of our verilator simulation
  - But also our top level hardware build
- Solution: Define separate top level ports

# AutoFPGA Config





What about the top level I/O's?

- Solution: Define separate top level ports

`@TOP.PORTLIST=` *List toplevel ports*

```
o_audio
```

`@TOP.IODECL=` *Declare our toplevel ports*

```
output wire     o_audio;
```

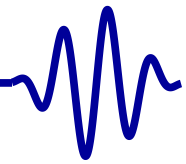
`@TOP.DEFNS=` *Define toplevel wires*

```
wire     w_audio_ce;
wire [15:0] w_sample;
```

`@TOP.MAIN=` *Connect these toplevel signals to main*

```
w_audio_ce, w_sample, o_audio
```

# AutoFPGA Makefile





Last steps:

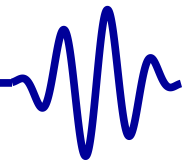
- Put your config, `tonegen.txt`, in the `autodata` directory
- Include `tonegen.txt` in the `autodata/Makefile`
- Place your logic, `tonegen.v` and `sintable.v`, into the `rtl` directory
- Re-run AutoFPGA, `make autodata`
- Rebuild the rest of the project

# Build problems?



If you struggle at all to get AutoFPGA to do what you want

- There's a -d option you can enable
- This will produce an `autofpga.dbg` file
- Shows you how the data is transformed throughout





# Formal Verification

# Bus Slaves

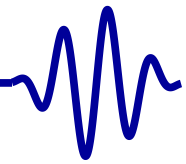


- Formally verifying signal processing blocks can be a challenge
  - We'll debug those via simulation
- Formally verifying bus slaves is easy
  - A bus slave failure will hang your entire design
  - Complicates debugging
  - Could leave you in FPGA Hell

*Always* formally verify any and all bus components

- The time to get into the habit starts now

# Property Files





The first thing you will need is a bus property file

- Here's a Wishbone slave property file
- Any core that passes this property check will not hang the bus
- That's not the same as proving that it will work
  - It just won't hang the bus

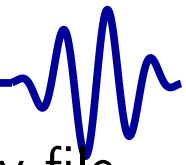
# Instantiation



Instantiate this core in a formal property section of `tonegen.v`

- You'll need to set some parameters
- `AW`=2, since we have $2 = \log_2 4$ registers
- `F_MAX_STALL`= 1, otherwise maximum stalls won't be checked
- `F_MAX_ACK_DELAY`= 2, since we return all results in one cycle
- `F_LGDEPTH`= 2, specifies that two-bit counters can be used to keep track of the number of outstanding bursts

# Instantiation



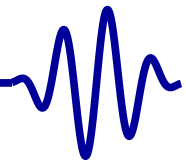


You'll also need to capture three results from the property file

- The number of total requests that have been made, `fwb_nreqs`
- The number of total acknowledgments received, `fwb_nacks`
- The total number of outstanding requests, `fwb_outstanding`

```
localparam      F_LGDEPTH=2;

wire [F_LGDEPTH-1:0] fwb_nreqs, fwb_nacks,
        fwb_outstanding;
```

# Instantiation

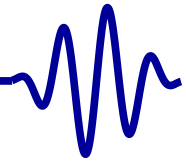




Your code should look something like . . .

```
localparam      F_LGDEPTH=2;

fwb_slave #(.AW(2), .F_LGDEPTH(F_LGDEPTH),
        .F_MAX_STALL(1),
        .F_MAX_ACK_DELAY(2))
fwb (i_clk, i_reset,
        i_wb_cyc, i_wb_stb, i_wb_we, i_wb_addr,
            i_wb_data, i_wb_sel,
        o_wb_ack, o_wb_stall, o_wb_data, 1'b0);
```

# Induction





To pass induction, you'll need just one more property

```
always @(*)
if (i_wb_cyc)
    assert(fwb_outstanding == (o_wb_ack ? 1:0));
```

Beware of the definition of `fwb_outstanding`

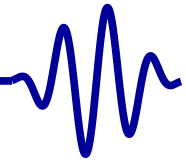
```
assign  f_outstanding = (i_wb_cyc)
                ? (f_nreqs - f_nacks):0;
```

- It will always be zero if `!i_wb_cyc`, regardless of `o_wb_ack`
- Hence the **if** `(i_wb_cyc)` above

# SymbiYosys script



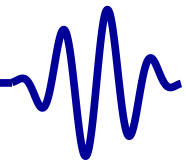
```
[options] # Place in bench/formal subdir
mode prove # Unbounded (i.e. induction) proof
depth 3

[engines]
smtbmc # The default engine

[script] # Yosys script
read -formal fwb_slave.v
read -formal sintable.v
read -formal tonegen.v
prep -top tonegen

[files] # Read files
fwb_slave.v
../../rtl/sintable.v
../../rtl/tonegen.v
```





# Simulation

# Simulation?



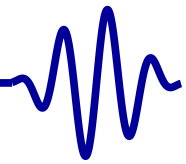


How will you know if it works

- Before you place it on hardware?
- . . . where you can no longer tell why it isn't working?

Simulation!

# Sim Script





As with your last design

- Build and run `sim/main_tb.cpp`

```
main_tb -d
```

- While running, run

```
wbregs frequency 0x0258bf25
wbregs amplitude 0x0300
```

- This should create a 440Hz tone
  - 440Hz is an “A” above middle C
  - Used extensively for tuning instruments

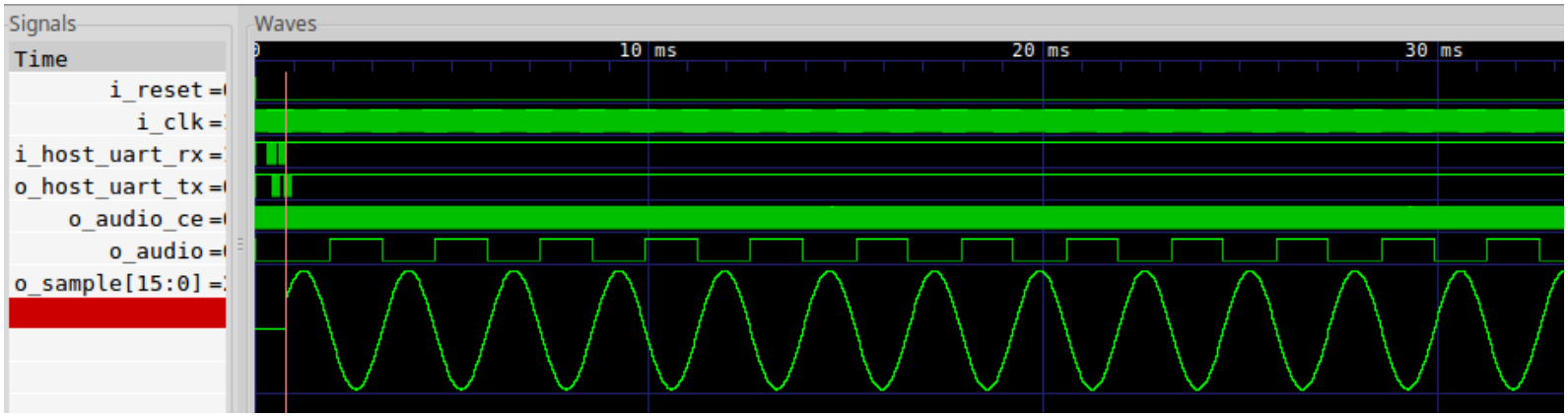
Let it run for several seconds, and then kill `main_tb` with Ctrl-C

- Open and examine the waveform

# VCD File



Using GTKWave, I produced this image

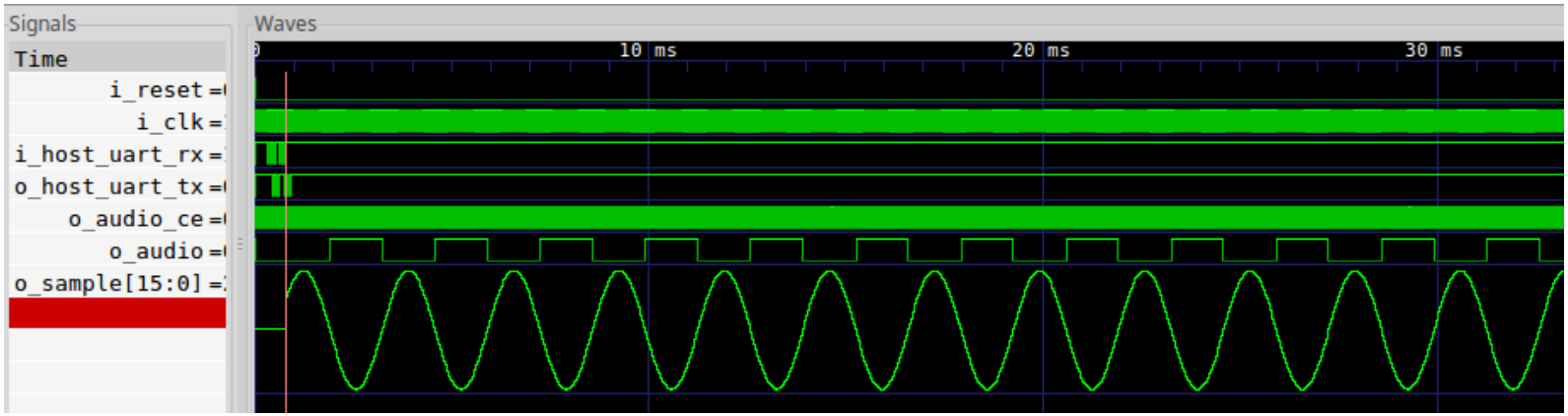



See if you can do it too

# VCD File



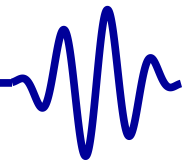
Using GTKWave, I produced this image




Notice the poor quality `o_audio`

- This was produced by using only the top bit of `o_sample`

# Data dump

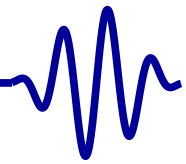




You should also have a `dumpfile.16t`

- Open it with Octave (or Matlab)
- How do the samples look?
-

# Data dump

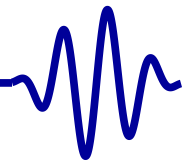




You should also have a `dumpfile.16t`

- Open it with Octave (or Matlab)
- How do the samples look?
- Try using the included `simscript.m` to display them

# Your turn!





Now modify your simulation to ...

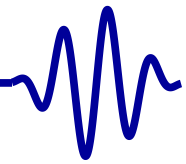
- Bring the `o_audio` output pin into Octave
- You'll need to filter it to get something recognizable

```
pin = % You'll need to set this
% Filter the incoming samples
fltrd = conv(ones(500,1), pin);
fltrd = conv(ones(500,1), fltrd);
fltrd = conv(ones(500,1), fltrd);
fltrd = conv(ones(500,1), fltrd);
fltrd = conv(ones(500,1), fltrd);
fltrd = conv(ones(500,1), fltrd);
plot(fltrd);
```

- How does the result look?
- It should look like `o_sample`

_

# Your turn!





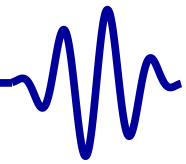
Now modify your simulation to ...

- Bring the `o_audio` output pin into Octave
- You'll need to filter it to get something recognizable

```
pin = % You'll need to set this
% Filter the incoming samples
fltrd = conv(ones(500,1), pin);
fltrd = conv(ones(500,1), fltrd);
fltrd = conv(ones(500,1), fltrd);
fltrd = conv(ones(500,1), fltrd);
fltrd = conv(ones(500,1), fltrd);
fltrd = conv(ones(500,1), fltrd);
plot(fltrd);
```

- How does the result look?
- It should look like `o_sample`
  - Does it?

# Your turn!





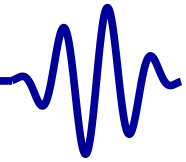
Now modify your simulation to ...

- Bring the `o_audio` output pin into Octave
- You'll need to filter it to get something recognizable

```
pin = % You'll need to set this
% Filter the incoming samples
fltrd = conv(ones(500,1), pin);
fltrd = conv(ones(500,1), fltrd);
fltrd = conv(ones(500,1), fltrd);
fltrd = conv(ones(500,1), fltrd);
fltrd = conv(ones(500,1), fltrd);
fltrd = conv(ones(500,1), fltrd);
plot(fltrd);
```

- How does the result look?
- It should look like `o_sample`
  - Does it? Can you plot the two together?

# Your turn!





Modify your design again so you can choose:

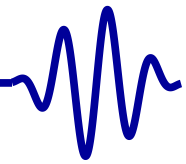
- Either the 1'bit audio output, or
- The PWM, PDM, or the Delta-Sigma output

Which is better?

- You may wish to look at the FFT of your tone

```
freq = (1:length(fltrd)) ./length(fltrd);
freq = (freq - 1/2) * sample_rate;
plot(freq,fftshift(abs(fft(fltrd))));
xlabel('Frequency (Hz)');
ylabel('Magnitude');
```

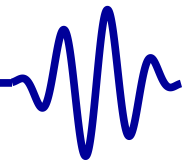
- Or compare the FFT's of each approach against the others
- Perhaps you want to adjust your design to create all four outputs at once





# Hardware

# Build it!





This is the moment you've been waiting for!

- It's now time to build your design for your board
- Connect the amplifier
- Is the sound at all what you expect?
- Which approach "sounds" the best?
- Can you create a script to play Yankee Doodle?