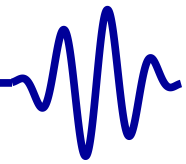
Gisselquist
Technology, LLC

4. Accessing On-Chip Memory

Daniel E. Gisselquist, Ph.D.

Lesson Overview



Objective: Building a bus access to on-chip memory

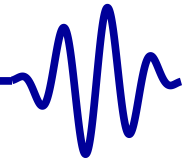
- On-chip memory is almost as easy as register access

Work in Progress

This lesson is currently a work in progress.

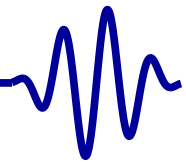
It will remain so until . . .

- I've built the design myself



Project

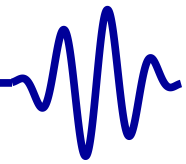
Waveform Generation



Let's build a project to output a waveform

- Initially, let's make one or more LEDs blink
- Using a custom pattern read from memory
  - We'll write a special blinking sequence to memory
  - Then read the blink pattern from memory
- At a programmable and controllable rate
  - A second/separate module will control frequency
  - Will also control: start, pause, repeat, and reset

Waveform Generation



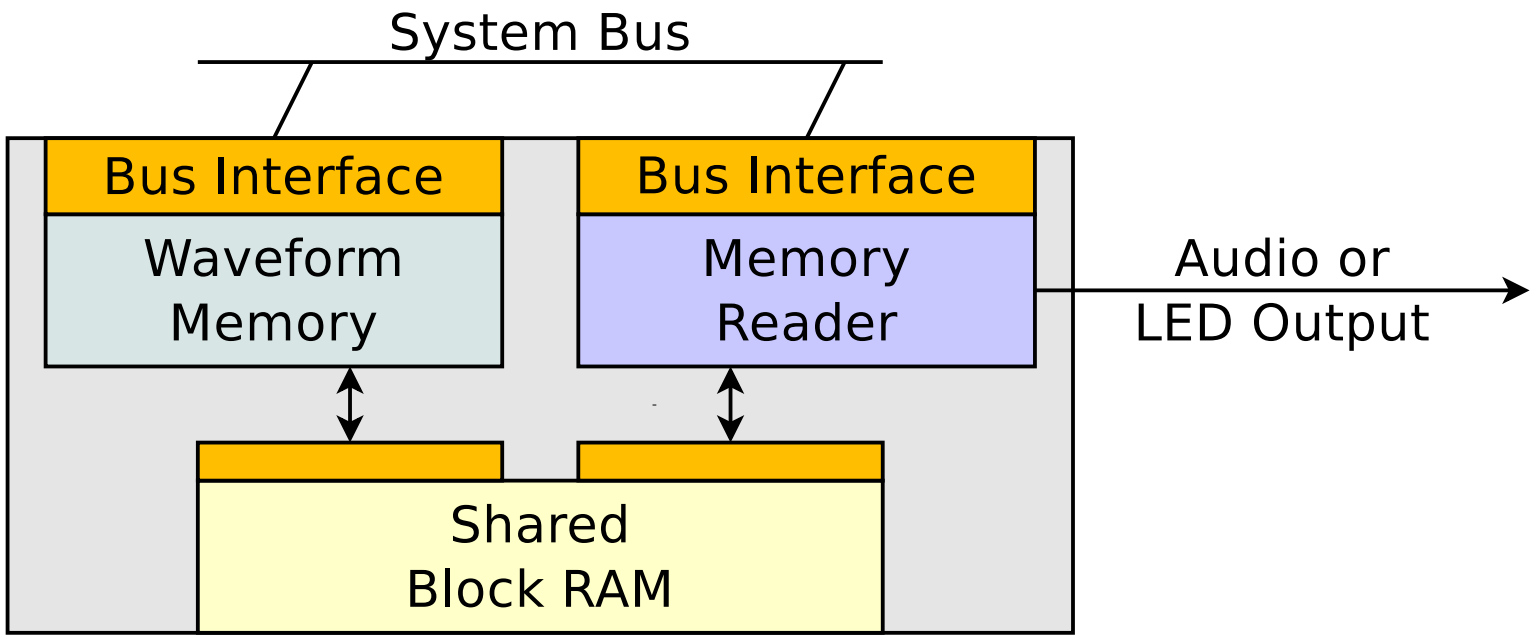
Let's build a project to output a waveform

- Initially, let's make one or more LEDs blink
- Using a custom pattern read from memory
- At a programmable and controllable rate
- We'll then move to an audio waveform generator
  - If you have an audio device, you can play sound here.
- Bonus: (for those so inclined)
  - You can also make any FPGA into an impromptu FM transmitter
  - Even without audio, therefore, you should be able to make an audio waveform using unintentional electromagnetically produced interference on an I/O pin.

Sound like a plan? Let's go!

Project Structure

The more complex a design gets, the more it helps to draw it out.

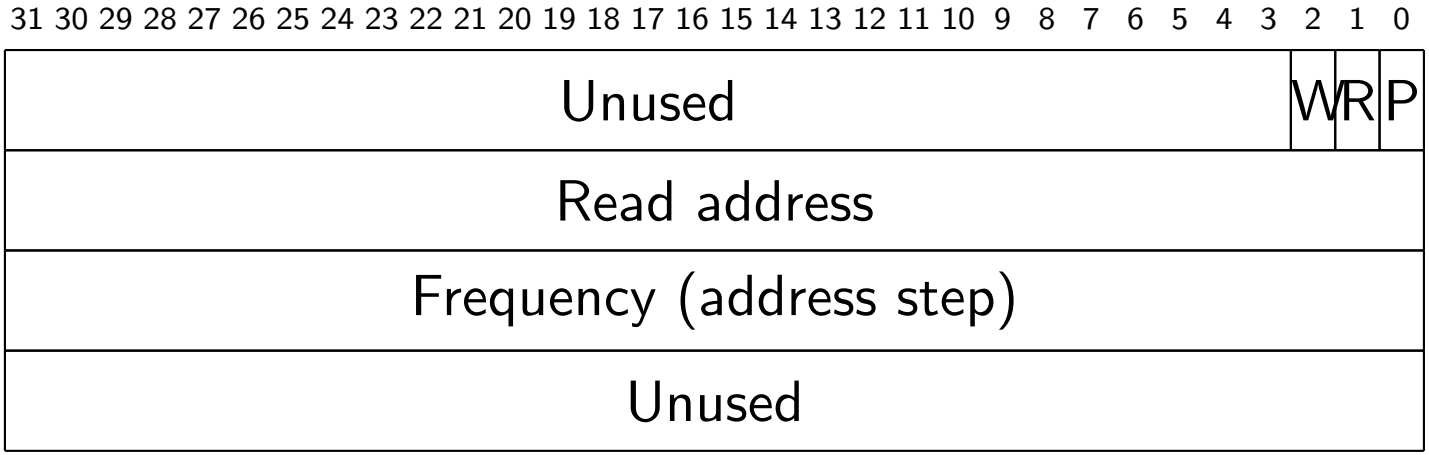


We'll need two bus interfaces

- One for waveform "memory"
- The other for output control

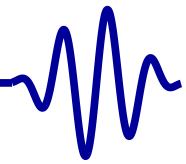
Control Requirements

We'll control our design with a couple of knobs:

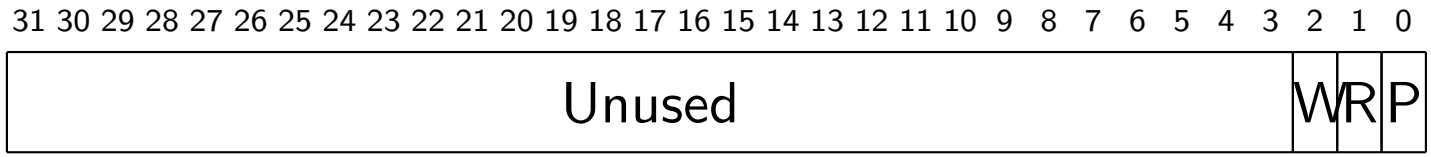


1. Playback control
2. Read address
3. Playback speed or frequency
4. That leaves one unused register

Control Requirements

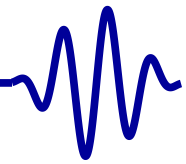


We'll control our design with a couple of knobs:



1. Playback control
   - One bit, P, controls playback (1) or pause (0)
   - Writing to another bit, R, resets the address
   - A third bit, W, controls whether we wrap from the end of memory or stop

Control Requirements



We'll control our design with a couple of knobs:

1. Playback control
2. Read address
   - Reads tell you where we are at in the cycle
   - Writes set up the next address
3. Playback speed
   - This will be our address increment
4. That leaves one register left
   - I'll leave it unused. You can do with it as you wish.

We'll come back to this in a bit.

- Let's discuss the block RAM component first

Memory Core

Memory is actually really easy. You know most of this already.

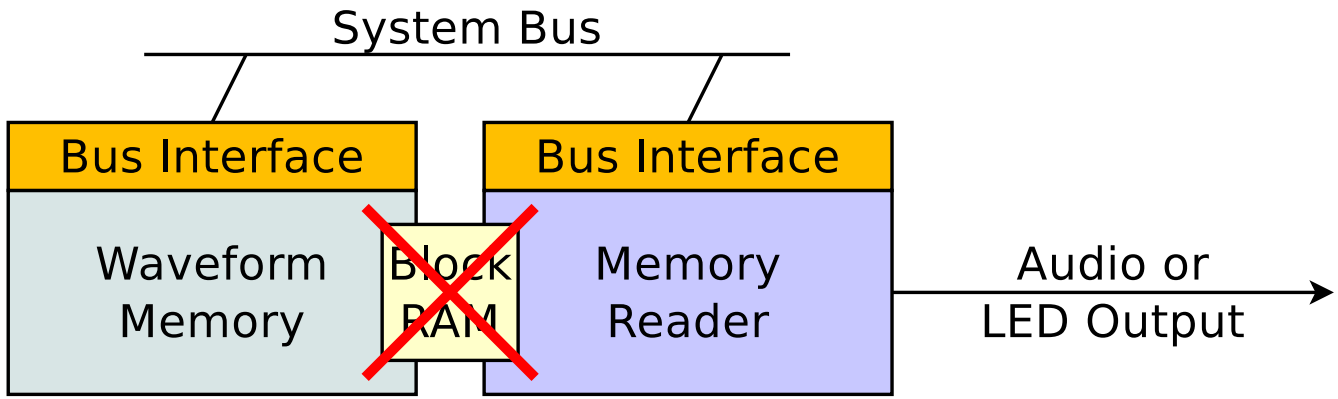
- Declare the memory

```
parameter W = 32, // Match the bus width
          LGNA = 5; // Log of the memory size

reg       [W-1:0] mem [0:(1<<LGNA)-1];
```

But how shall two bus interfaces share the same memory?

Memory Core

Full Block RAMs can't be passed directly from one module to the next



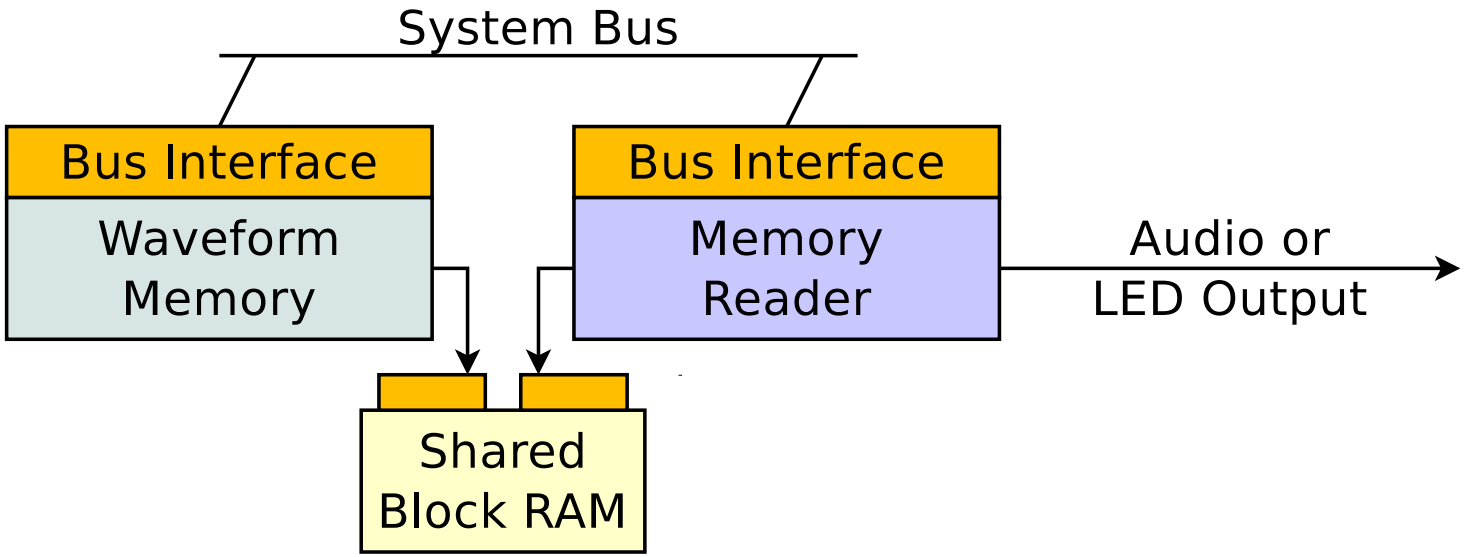
The following code *doesn't* work

```
input    wire     [W-1:0] mem [0:(1<<LGNA)-1];
```

Memory Core

Full Block RAMs can't be passed directly from one module to the next

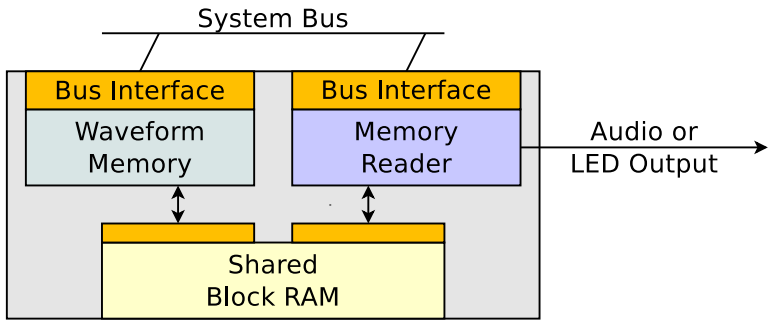
- We could create an interface for this memory for the control core: `Rd`, `Addr`, `Rdata`, etc.



Memory Core

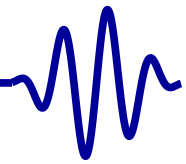
Full Block RAMs can't be passed directly from one module to the next

- We could create an interface for this memory for the control core: `Rd`, `Addr`, `Rdata`, etc.
- Or we can place both cores in the same module
  - The bus interfaces could be shared
  - Or the two interfaces could be separate



Today, we'll place both interfaces together in the same module

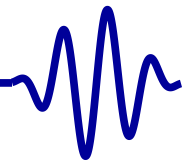
Memory Core



To handle two bus ports,

- I'll prefix one with `i_mem_*` or `o_mem_*`
  - rather than `i_wb_*` or `o_wb_*`
- The other with `i_ctrl_*` or `o_ctrl_*`
- I'll drop these prefixes in these slides, just because screen space is tight
  - You'll see these full names in the example files

Memory Core

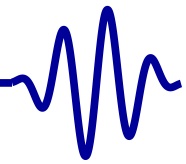


Let's continue building our memory module

- We've already declared our memory
- Now we'll handle memory writes

```
always @(posedge i_clk)
if (i_stb && i_we)
begin
    if (i_sel[3])
        mem[i_addr][31:24] <= i_data[31:24];
    if (i_sel[2])
        mem[i_addr][23:16] <= i_data[23:16];
    if (i_sel[1])
        mem[i_addr][15: 8] <= i_data[15: 8];
    if (i_sel[0])
        mem[i_addr][ 7: 0] <= i_data[ 7: 0];
end
```

Memory Core



Let's continue building our memory module

- We've already declared our memory
- Now we'll handle memory writes
- Then memory reads

```
always @(posedge i_clk)
        o_data <= mem[i_addr];
```

- Wishbone signaling

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

initial o_ack <= 0;
always @(posedge i_clk)
        o_ack <= i_stb;
```

Memory Core

Memory is actually really easy. You know most of this already.

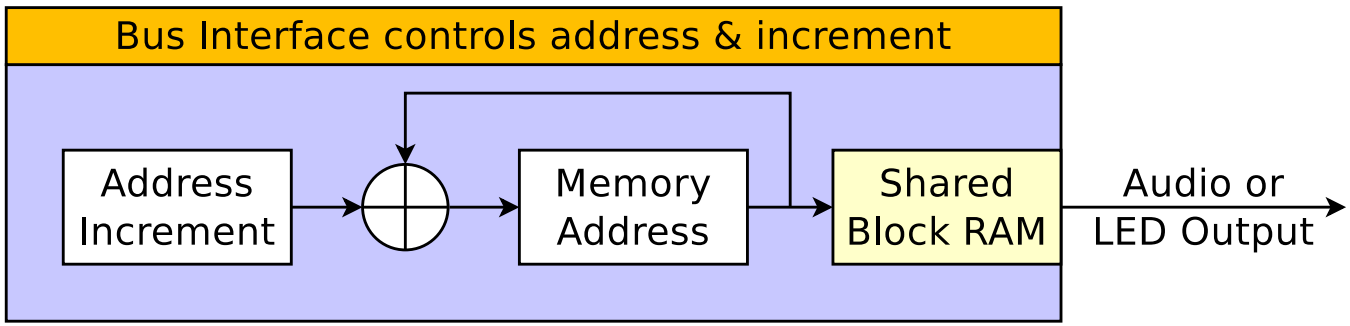
- Declare the memory
- Handle memory writes
- Handle memory reads
- Handle Wishbone signaling

We'll use a different formal technique to verify this

- We'll come back to that in a moment

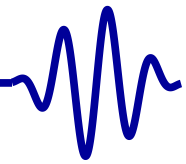
Control Core

Now let's build our control core



- It'll just read from an incrementing address through memory

Control Core



Now let's build our control core

- We'll need a memory address

```
// Note that at 32-bits, this address has many
// bits than the LGNA bits our memory requires
reg     [31:0]  memaddr;
```

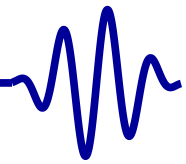
- From this we'll read from memory

```
always @(posedge i_clk)
        memword <= mem[memaddr[31:32-LGNA]];
```

What if we don't want to read 32-bits at a time?

- Sorry, all memory reads are the full width
- To read 8-bits at a time, we'll need to select our 8-bits from among these 32-bits

Control Core



Now let's build our control core

- We'll need a memory address
  - The bottom 2-bits will tell us which octet of our word
- We'll read with 2-bits less in the address

```
always @(posedge i_clk)
        memword <= mem[memaddr[31:30-LGNA]];
```

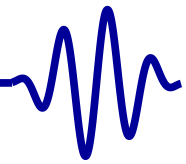
- We also need to record the sub address bits

```
always @(posedge i_clk)
        subaddr <= memaddr[29-LGNA:32-LGNA];
```

- On the next cycle, we can get the 8-bits we want

```
always @(posedge i_clk)
        o_sample <= memword >> (subaddr * 8);
```

Control Core



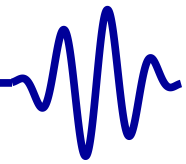
Now let's build our control core.

- Bus writes

1. First the control address

```
always @(posedge i_clk)
if (i_stb && i_we&& i_addr == 0 && i_sel[0])
begin
        playing    <= i_data[0];
        reset_addr <= i_data[1];
        wrap       <= i_data[2];
end else
        // Clear the reset request on
        // the next clock
        reset_addr <= 1'b0;
```

Control Core

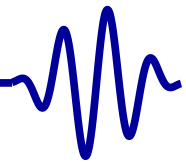


Now let's build our control core.

- Bus writes
  1. First the control address
  2. Then the memory address
     - Here we'll need to do a couple of things
     - First, read any new value from the bus

```
always @(posedge i_clk)
if (i_stb && i_we && i_addr == 1)
        // Q: How would you handle i_sel here?
        memaddr <= i_data;
else // ...
```

Control Core

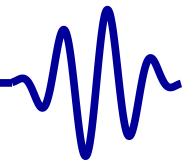


Now let's build our control core.

- Bus writes
  1. First the control address
  2. Then the memory address
     - Here we'll need to do a couple of things
     - First, read any new value from the bus
     - Reset our address if requested

```
// ...
else if (reset_addr)
        memaddr <= 0;
else // ...
```

Control Core



Now let's build our control core.

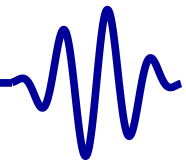
- Bus writes
  1. First the control address
  2. Then the memory address
     - ...Update the address if we are currently "playing"

```
// ...
else if (playing)
        // Step through memory
        memaddr <= memaddr + speed;
```

Note that I haven't implemented the play only once feature

- I'll leave that control bit to you
- You'll want to reset `memaddr` and `playing` on any overflow, if you aren't continuously playing

Control Core



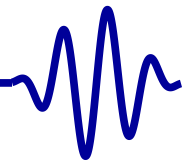
Now let's build our control core.

- We'll need a memory address
- From this we'll read from memory
- Now let's handle bus writes

1. First the control address
2. Then the memory address
3. Finally the speed (a.k.a. increment) register

```
always @(posedge i_clk)
if (i_stb && i_we && i_addr == 2)
        // Q: How would you handle i_sel here?
        speed <= i_data;
```

Control Core

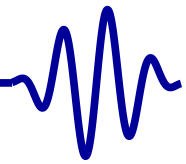


Now let's build our control core.

- We'll need a memory address
- From this we'll read from memory
- Now let's handle bus reads

```
wire [31:0]     w_control_data;

assign  w_control_data = { 29'h0,
                wrap, 1'b0, playing };
always @(posedge i_clk)
case(i_addr)
0: o_data <= w_control_data;
1: o_data <= memaddr;
2: o_data <= speed;
2: o_data <= 0;
endcase
```

Control Core



Now let's build our control core.

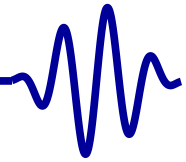
- We'll need a memory address
- From this we'll read from memory
- Now let's handle bus reads

```
wire [31:0]     w_control_data;

assign  w_control_data = { 29'h0,
                wrap, 1'b0, playing };
```

This is a common form I use often

- Declare a bus-wide register, here it is `w_control_data`
- Assign it as appropriate, then return it on any read
- Simulation tools will then leave the register in the trace so you can examine it as desired

Control Core



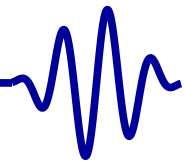
Now let's build our control core.

- We'll need a memory address
- From this we'll read from memory
- Now let's handle bus reads, and
- The last of the bus odds and ends

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

initial o_ack = 1'b0;
always @(posedge i_clk)
        o_ack <= i_stb;
```

Simple enough?

Resets?

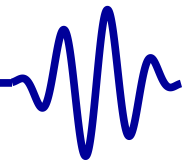


We haven't included resets into our design

- Do we need them?

Your thoughts?

Resets?

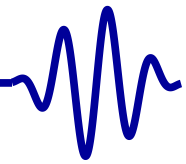


Do we need a reset? Here are my thoughts:

- FPGAs support **initial** statements
  - ASICs do not support **initial** statements
  - FPGA support is (usually) pretty good
- LED or audio glitches will never be noticed
- We will still need to reset anything bus related
  - This is primarily the `o_ack` register

```
initial o_ack = 1'b0;
always @(posedge i_clk)
if (i_reset)
        o_ack <= 1'b0;
else
        o_ack <= i_stb;
```

  - We could also reset `playing` and `memaddr`



Formal Verification

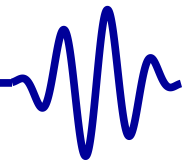
Property Files

Verifying bus components always starts with a bus property file

- If you are struggling to know where to start, this is it
  - You can do a lot more, but this is often a good first start
- Here's my Wishbone slave property file
  - You should also find a copy in the exercise files
  - I have other bus property files as well
- Any core that passes this property check will obey the bus protocol
  - Core's that don't, might hang your design
  - Hung designs are hard to debug, and can lead to endless frustration

Don't get stuck: always start with the property file for your bus

Property Files



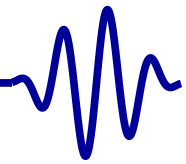
Checking the memory bus interface

- All we need to adjust is the address width of the checker

```
fwb_slave #(
        // Select the address width on
        // instantiation
        .AW(LGNA-2),
        .F_LGDEPTH(F_LGDEPTH),
        .F_MAX_STALL(1),
        .F_MAX_ACK_DELAY(2))
fmem (i_clk, i_reset,
        i_mem_cyc, i_mem_stb, i_mem_we,
          i_mem_addr, i_mem_data, i_mem_sel,
        o_mem_ack, o_mem_stall, o_mem_data, 1'b0,
        fmem_nreqs, fmem_nacks, fmem_outstanding);
```

Don't forget to check both bus interfaces!

Property Files



Bus property files only check *bus properties*, like . . .

- One and only one acknowledgment per request
- Stalled requests actually stall, etc.

The don't check whether or not the core . . .

- Does the "right thing" on any writes
- Returns the "right data" on any reads
- Does the "right thing" in the rest of the logic

That leaves us with some other things to check

Verifying memory

Verifying memory requires a different formal verification approach

- We discussed this in the beginner's tutorial

1. Pick an arbitrary address

```
(* anyconst *)  reg [LGNA-3:0]  f_addr;
```

2. And a value for that address

```
reg [LGNA-3:0]  f_data;

initial assume(f_data == mem[f_addr]);
```

Verifying memory

Verifying memory requires a different formal verification approach

1. Pick an arbitrary address
2. And a value for that address
3. Update that value on any bus write

```
always @(posedge i_clk)
if (i_stb && i_we && i_addr == f_addr)
        f_data <= i_data;
```

4. Assert that our sampled value matches actual memory

```
always @(posedge i_clk)
        assert(f_data == mem[f_addr]);
```

Verifying memory

Verifying memory requires a different formal verification approach

1. Pick an arbitrary address
2. And a value for that address
3. Update that value on any bus write
4. Assert that our sampled value matches actual memory
5. Assert that bus reads actually return the right answer

```
always @(posedge i_clk)
if (f_past_valid && !$past(i_reset)
        && $past(i_wb_stb && i_addr == f_addr))
begin
        assert(o_data == f_data);
        assert(o_ack);
end
```

Verifying memory

Verifying memory requires a different formal verification approach

1. Pick an arbitrary address
2. And a value for that address
3. Update that value on any bus write
4. Assert that our sampled value matches actual memory
5. Assert that bus reads actually return the right answer

You'll need to adjust the write slightly to handle write strobes

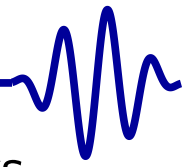
- This should be straightforward

Verifying the Control Port

You should be able to come up with some properties to verify the control port

- Start with the bus property file, then . . .
- Verify that if ever `wrap` is false, then `memaddr` doesn't wrap without stopping
- Pick an address. Verify the output sample is correct if that address is ever selected
  - You'll need to follow the address through our pipeline
  - From memory read to final output sample
- Verify that the memory increments at `speed` units per clock
  - This is trickier than it sounds–watch out for overflow!

Measuring Throughput



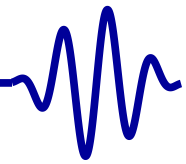
Measuring bus throughput is really easy using SymbiYosys

- We can use **cover**() for this purpose
- We just need to **cover**() some number of returns
  - Let's generate a trace showing four returns

```
always @(posedge i_clk)
if (i_reset)
        cvr_returns <= 0;
else if (o_ack)
        cvr_returns <= cvr_returns + 1;

always @(*)
        cover(cvr_returns == 4);
```

Let's go ahead and try this.

SymbiYosys script



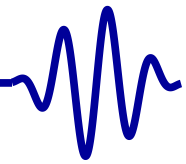
Don't forget, to use **cover**() you'll want to use tasks

- Let's create tasks `prf` and `cvr`
- `prf` will run a normal proof
- `cvr` will run a cover check

This should look just like we did it in the beginner's tutorial

```
[tasks]
prf
cvr

[options]
prf: mode prove
cvr: mode cover

## Rest continues as before
```

SymbiYosys and Make



I really like using make with my SymbiYosys runs

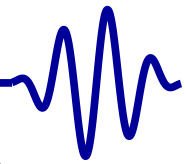
- The PASS file makes a good make target

```
RTL := ../../rtl
DEPS:= $(RTL)/wavegen.v fwb_slave.v
wavegen_prf/PASS: $(DEPS) wavegen.sby
        sby -f wavegen.sby prf
wavegen_cvr/PASS: $(DEPS) wavegen.sby
        sby -f wavegen.sby cvr
```

Now if ever our file ever changes, make will catch it

- make can automatically rebuild proofs across a project
- Can quickly let you know if something changed significantly
- Only ever as good as the properties you write

Do not pass Go



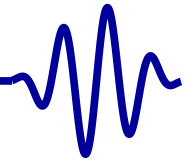
The next step is integrating this core into a bigger design

- *Do not proceed to integration until you know your core works!*

Take whatever time you need get it your core to pass

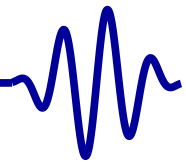
- This applies especially to your bus interfaces
- Do what you can with the rest
- If you miss a bug later, then adjust your properties to catch it next time and come back here and re-do this step

Debugging only gets harder from here on out



AutoFPGA

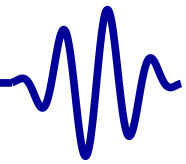
Bus connection



Now that we have a core to play with, let's wire it up!

- As before, we can use AutoFPGA to connect this to the bus
- Only really one different/unique thing about this core
  - It has two bus interfaces
- We'll handle this by telling AutoFPGA we have two cores
  - Using only one configuration file
  - We'll then reference the bus connections of the one from within the other

Memory Config



We'll start with the memory configuration

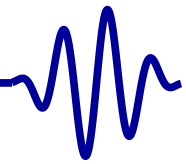
```
@PREFIX=wavegen
@SLAVE.BUS=wb              Connect to bus named wb
@SLAVE.TYPE=DOUBLE         Respond in one cycle
@MAIN.PORTLIST=            Output LEDs
```

```
        o_led
```

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

```
        output wire [7:0] o_led;
```

Memory Config



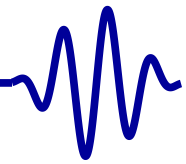
We'll want to make the memory size adjustable

- AutoFPGA supports tags with integer values
- These start with `@$` `@$LGNADDR=4` 2^4 *or* 16 *words*
- We can then use this value to calculate the number of bus words this core supports

  `@$NADDR=(1<<@$(LGNADDR))` *Num bus words*

  – Note that the initial `@$` is critical
  – It keeps this value from being interpreted as a string

Main insert



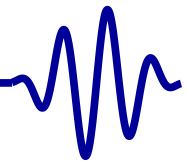
The next step is to insert this core into our main design

`MAIN.INSERT=` *This will be copied into main.v*

```
wavegen #(.LGNA(@$(LGNADDR)+2))
@$(PREFIX)i(i_clk, i_reset,
        //
        // The port list for the memory port
        @$(SLAVE.PORTLIST),
        //
        // Grab the port list for the control port
        @$(wavectrl.SLAVE.PORTLIST),
        //
        // Our 8-bit output, headed to the LEDs
        o_led);
```

Let's step through this

Main insert

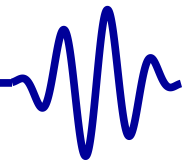


The next step is to insert this core into our main design

- The core has an `LGNA` parameter determining memory size
- The bus also needs to know this memory size
- Software accessing this core also needs to know this memory size
- We defined this size using `@$LGNADDR` above
- We can now reference it as `@$(LGNADDR)`
- AutoFPGA will substitute this with our calculated value

```
wavegen #(.LGNA(@$(LGNADDR)+2))
// ...
```

Main insert

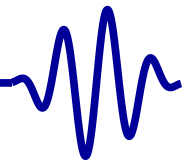


The next step is to insert this core into our main design

- We need to give our core a name
- I often use `@$(PREFIX)i` for this
- In this case, that expands to `wavegeni`
- Software accessing this core also needs to know this memory size
- We defined this size using `@$LGNADDR` above
- We can now reference it as `@$(LGNADDR)`
- AutoFPGA will substitute this with our calculated value

```
wavegen #(.LGNA(@$(LGNADDR)+2))
@$(PREFIX)i(i_clk, i_reset),
        // ...
```

Main insert

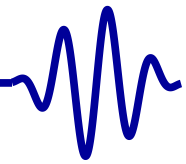


The next step is to insert this core into our main design

- Connecting to a bus involves a lot of port connections
- AutoFPGA provides `@$(SLAVE.PORTLIST)` to make this easier
  - This automatically adjusts for the right address width
  - It does require positional argument assignment
  - Your code needs to match
- AutoFPGA also defines `@$(SLAVE.ANSIPORTLIST)` if you want to use named ports instead
  - These names can be parameterized to match your design
  - All defined bus connections are still required

```
        @$(SLAVE.PORTLIST),
```

Main insert



The next step is to insert this core into our main design

- What about our other bus connection?
- We'll define the control interface with a `@PREFIX` of `wavectrl`
- We can reference it from within this core's set up
  - Tags not defined in the current context are searched for in the next context up
  - That's where we'll find `wavectrl.SLAVE.PORTLIST`

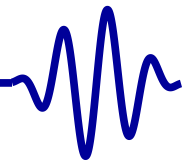
```
        @$(wavectrl.SLAVE.PORTLIST),
```

The last piece, `o_led` are just the LED driving wires

```
        o_led);
```

This all gets copied into `main.v`

Register Address



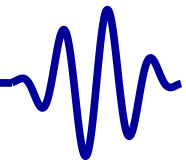
We'll also want to know the ultimate address of our core

- This is determined by AutoFPGA
  - It's used internally to configure the interconnect
  - We also want this value in several output files

```
@REGS.N=1                          One named address
@REGS.0=0 R_WAVEFORM WAVEFORM      Defines names in regdefs.*
@REGDEFS.H.INSERT=                 Put the length into regdefs.h
```

```
// Size of our waveform memory
#define WAVELEN (1<<@(LGNADDR))
```

CPU Header



We'll also want to know the ultimate address of our core

- We might want also to know this address within a soft-core CPU
- The following will be copied into a `board.h` file

  `@BDEF.OSVAL=` *Define our memory's base address*

```
static volatile unsigned *const @$(PREFIX)
        = ((unsigned *)@$[0x%08x](REGBASE));
```

- `@$REGBASE` is the base address of this component
- The [0x%08x] notation just specifies how this address is to be formatted: Eight hex digits following a `0x` prefix
- It follows the C/C++ printf conventions

Simulation

Let's create a simple simulation script to **printf()** any time our output changes

`@SIM.CLOCK=clk` *Define the relevant clock*

`@SIM.DEFNS=` *Define a local C++ variable*

```
int m_last_led;
```

`@SIM.INIT=` *Initialize it*

```
        m_last_led = 0;
```

Simulation

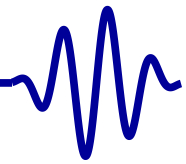
Let's create a simple simulation script to **printf()** any time our output changes

- Now, what shall we do on every clock tick?
- That's provided by the `@SIM.TICK` tag
- The field gets copied into our `main_tb.cpp` file

`@SIM.TICK=` *Do this on each clock tick*

```
if (m_core->o_led != m_last_led) {
        m_last_led = m_core->o_led;
        printf("LED output: %02\n",
                m_last_led);
}
```

Control Registers



We still need to define our control interface and registers

- For this, we'll place a second `@PREFIX` tag in the config file

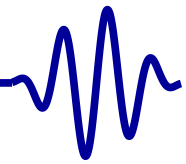
```
@PREFIX=wavectrl        Define a second bus interface
@NADDR=4                With four registers
@SLAVE.BUS=wb           Connecting to wb bus
@SLAVE.TYPE=DOUBLE      Taking one clock
@REGS.N=3               Having three named registers
```

- Now, let's define our three registers

```
REGS.0=0 R_WAVECTRL WAVECTRL   Control register
REGS.1=1 R_WAVEADDR WAVEADDR   Current address
REGS.2=2 R_WAVEFREQ WAVEFREQ   Step reg
```

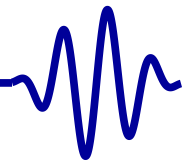
Remember, we covered the `REGS.#` format in the last lesson

Running AutoFPGA



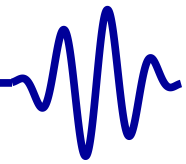
Now, when you run AutoFPGA, you'll get ...

- `main.v` containing bus connection logic, crossbar references, etc.
- `main_tb.cpp` containing your component's simulation logic
- `regdefs.h` defining C++ names for your registers
  - We chose to prefix all of these with `R_`
  - These are turned into **#define** statements, assigning the associated name with its address
- `regdefs.cpp` defining a name to be used with `wbregs`
- `board.h` defining your registers



Simulation

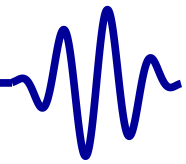
Simulation



Always test your designs in simulation before hardware

- Most of the simulation files are already built for you
- You'll still need to compile them
- Once done,
  1. Run `main_tb` in one window
     - You can use `-d` to create a VCD trace as well
  2. Use wbregs to interact with your design
     - It should recognize the names `WAVECTRL`, `WAVEADDR`, `WAVEFREQ`, and `WAVEFORM`

Test Sequence



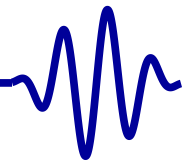
Try creating a test sequence for your simulation

- We have 16 slots to play with
- How about blinking three times, . . .
- While scrolling an LED back and forth?
- Hint: You'll need to look up the `WAVEFORM` address

Assuming the waveform address is 0x0400

```
wbregs 0x400 0x11 # Turn LED[0] on
wbregs 0x404 0x10 # LED[0] off
wbregs 0x408 0x21 # LED[0] on
wbregs 0x40c 0x20 # LED[0] off
wbregs 0x410 0x41 # LED[0] on
wbregs 0x414 0x40 # LED[0] now stays off
wbregs 0x418 0x80 # LEDs[7:4] keep scrolling
wbregs 0x41c 0x80
wbregs 0x420 0x40
## etc.
```

Test Sequence



Try creating a test sequence for your simulation

- We have 16 slots to play with
- How about blinking three times, . . .
- While scrolling an LED back and forth?
- Hint: You'll need to look up the `WAVEFORM` address

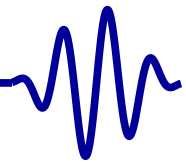
Assuming the waveform address is `0x0400`

```
wbregs 0x400 0x11 # Turn LED[0] on
## ...
```

Did it work?

- If not, why not?
- Remember: I've been known to leave bugs behind for you to find and fix

Sim Control



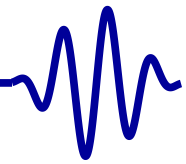
Now let's make that sequence blink

```
wbregs WAVECTRL 2 # Turn off, reset address
wbregs WAVEFREQ 0x10c70 # Some random frequency
wbregs WAVEFREQ 1 # Start it "playing"
```

Remember,

- You can increase `WAVEFREQ` to make the sequence go faster
- Or decrease `WAVEFREQ` to make it go slower

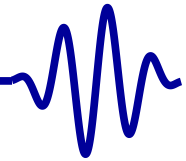
Caution!



One warning:

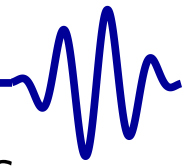
- If you are generating a VCD file
- It can get very big quickly
- Kill the simulation with a Ctrl-C before it gets too big

Have fun!



Host Control

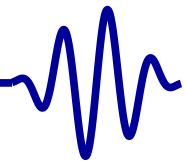
Host control



You should be able to generate a C++ file to control this sequence

- Make the LED's blink 1-8 times based on a C++ program
- Start with `wbregs` as an example to work from

LED blinks



Here's an example control program snippet

```
blinks = atoi(argv[1]);
for(int k=0; k<WAVELEN; k++)
        m_fpga->writeio(R_WAVEFORM+k*4, 0);
for(int k=0; k<blinks
                && k<WAVELEN/2; k++)
        m_fpga->writeio(R_WAVEFORM+k*8, 1);
```

LED blinks

Sometimes you can go faster by writing all of the values at once

- Really depends upon the debugging bus implementation
- Ours defines a `writei()` method we can use

```
class    DEVBUS {
public:
        typedef uint32  BUSW;
        // ...
        virtual void     writei(
                const BUSW address,
                const int len,
                const BUSW *buf) = 0;
```

- If you don't tell the debugging bus that you want to send a lot of data then it can't optimize the transfer

How would our C++ software change if we used `writei()`?

LED blinks

Here's an example C++ program to transfer our data using **writei**()

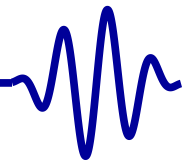
```
int tbl[16];

for(int k=0; k<WAVELEN; k++)
        tbl[k] = 0;

for(int k=0; k<blinks && k<WAVELEN/2; k++)
        tbl[k*8] = 1;

m_fpga->writei(R_WAVEFORM, WAVELEN, tbl);
```

That's cool, but might we do some better?

- Can you make your LED blink 17 times with a 16-long memory?

Not fun enough. Let's do even more.

Audio

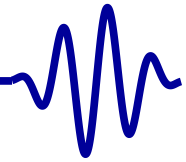


We could easily ...

- Adjust our wavelength size to 1024 or more values
- Rename `o_led` to something more appropriate for audio, like `o_sample`
- Create a sinewave table

```
char    sbuf[WAVELEN*4];

for(int k=0; k<WAVELEN*4; k++) {
        int     sample;
        sample = (127)*sin(2.0*M_PI*k
                / (double)WAVELEN/4.);
        sbuf[k] = (char)sample;
} m_fpga->writei(R_WAVEFORM, WAVELEN, sbuf);
```

Audio



If the table holds one wavelength, . . .

- Pitch follows from `WAVEFREQ` and your system clock rate, f_{SYS} Hz

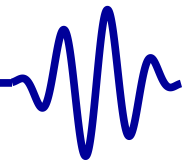
$$f = \frac{\texttt{WAVEFREQ}}{2^{32}} f_{\text{SYS}}$$

- That gives us a formula for `WAVEFREQ`

$$\texttt{WAVEFREQ} = \frac{f_{\text{DESIRED}}}{f_{\text{SYS}}} 2^{32}$$

- Note that the pitches are evenly spaced in `WAVEFREQ`
- That gives `WAVEFREQ` units of frequency–just not Hz

Audio

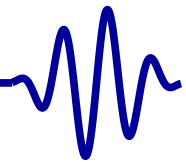


Here are some pitches you might be interested in

Musical note	Frequency (Hz)	WAVEFREQ
Middle C	261.6256	0x002be5
D	293.6648	0x003145
E	329.6276	0x00374d
F	349.2282	0x003a97
G	391.9954	0x0041c4
A	440.0000	0x0049d2
B	493.8833	0x0052dc

- The `WAVEFREQ` value was calculated assuming an f_{SYS} frequency of 100 MHz
- You can find more pitches on Wikipedia

Sim Challenges

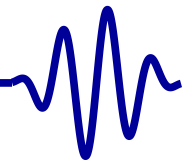


Yes, you can simulate this audio design

- How will you know if you got the pitch right?
- How big does the VCD file need to be to guarantee you achieved 440 cycles in one second?
- It might make more sense to just check a couple pitches
  - Consider only checking two cycles of any wavelength
  - You might also check against pitches above audio range
  - These will be easier to verify in simulation
- For example: Try adjusting `WAVEFREQ` to double the frequency
  - Did it double as expected?

We really need to move to hardware though.

Common Problems

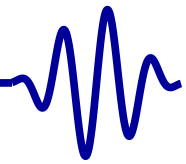


Many students struggle with assignments like this

- Sending a file of data to an FPGA's memory is a common problem
  - Where to start?
- Students will often start by attaching an on-board CPU
- When they run the design, it doesn't work
- They then have no idea why not
  - Was the CPU at fault?
  - How to get the CPU instructions into memory?
  - Was the software broken?
- How would you ever debug this?

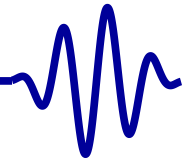
We're using a different approach entirely

Common Problems



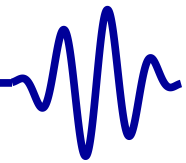
This approach is unique:

- You can simulate it entirely without a proprietary tool chain
  - Because everything is open
  - You can debug any failing part
  - You can trace the failure through parts not your own if necessary
- This simulation includes sending data to the design
  - This file can be arbitrary
  - Want to send your favorite song?
  - You'll be limited by the size of the RAM on your hardware
- And then verifying that the design properly works with the information we've sent it
  - You did get it to work, right?



Hardware

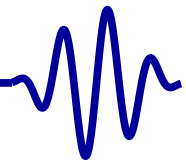
Build it!



You should be able to make your LED(s) blink in any sequence

- Modify the demo design for the number of LEDs you have
- Can you make the LED blink once, twice, four times in a row?
- Can you make it blink faster or slower?
- What frequency "speed" will cause your LED to blink at exactly 1Hz?
  - Can you make your LED blink at 2Hz using the same `WAVEFREQ` value?

Upgrades



- Our demo doesn't use the select lines
  - How would you modify it to use the select lines?
- Can you modify this to create an audio waveform?
  - A tone? Game sounds? Different instruments?
  - How about arbitrary waveforms: Speech, or music?
  - A SONAR pulse? (A tone with a duty cycle)
- Can you modify this to "transmit" on FM?
  - Perhaps this project will inspire you
  - More outputs "transmitting" will increase the signal strength
  - Longer wires from FPGA to output will help to match the "antenna" for better performance

FM Transmitter

I mentioned you could make an (unintentional) FM transmitter

```
reg [31:0] nco_phase, nco_step;

// Set this according to the FM frequency
// you want to transmit on
always @(posedge i_clk)
        nco_step = ?;

always @(posedge i_clk)
    nco_phase <= nco_phase + nco_step
        // Sign extend and scale our sample
        + {(8){sample[7]}, sample, 16'h0 };

always @(posedge i_clk)
        o_fm <= nco_step[31];
```

- Now send `o_fm` to an antenna

FM Transmitter

I mentioned you could make an (unintentional) FM transmitter

- I need an antenna?
  - For "best performance", yes
  - I was able to get it to transmit short distances with only unconnected FPGA outputs
  - For greater distance, I used all of my FPGAs outputs
    - This got me to 12-18 inches or so
    - My board was only a half inch wide.
    - A bigger board, with longer I/O traces should've been better "matched" to the half wavelength of the frequency I was working at (about a meter)
- Feel free to check out this example for more information