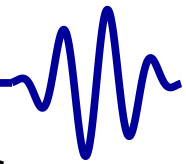
1. Blinky

Gisselquist
Technology, LLC

Daniel E. Gisselquist, Ph.D.

Lesson Overview



▷ Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Hardware Emulation
Host Software

Objective: To learn and become familiar with using a bus

- We'll use Wishbone in this lesson
  - Look for an AXI-lite example in the next lesson, and any accompanying appendices
- Build a basic general purpose output controller
- Extend it to handle inputs

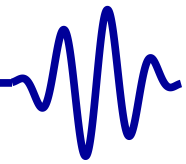
Work in Progress

▻ Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Hardware Emulation
Host Software

This lesson is currently a work in progress.

It will remain so until . . .

- I have a verified the instructions and
- Used them to generate a working example on my own
- I have some AXI-lite examples in an appendix



Lesson Overview

AutoFPGA Tools

Example AutoFPGA

Running the Demo

PWRCOUNT

RAWREG

In Hardware

What just happened?

What just happened?

The Baud Rate

The Buttons

Reconfiguration

GPIO

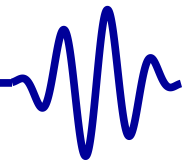
AutoFPGA

Hardware Emulation

Host Software

AutoFPGA Tools

Example AutoFPGA



Lesson Overview

AutoFPGA Tools

Example AutoFPGA

Running the Demo

PWRCOUNT

RAWREG

In Hardware What just happened?

What just happened?

The Baud Rate

The Buttons

Reconfiguration

GPIO

AutoFPGA

Hardware Emulation

Host Software

Let's learn our way around an AutoFPGA design

1. Clone and build AutoFPGA

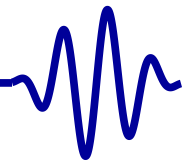
```
% git clone \
https://github.com/ZipCPU/autofpga
% cd autofpga
% make
% export PATH=$PATH:<path_to_autofpga>/sw
```

2. Clone and build AutoFPGA's demo

```
% git clone --recurse-submodules \
https://github.com/ZipCPU/autofpga-demo
% cd autofpga-demo
% make
```

Let's take a look around the AutoFPGA-demo

Running the Demo



Lesson Overview
AutoFPGA Tools
Example AutoFPGA
Running the Demo
PWRCOUNT
RAWREG
In Hardware
What just happened?
What just happened?
The Baud Rate
The Buttons
Reconfiguration
GPIO
AutoFPGA
Hardware Emulation
Host Software

Running the demo requires a couple of steps

1. Open a terminal, and start the Verilator simulation

```
% cd <path_to_autofpga_demo>/sim
% ./main_tb
Listening on port 9467
```

2. You can kill this simulation with Ctrl-C when you are done
3. In a second terminal, let's read a register from the sim

```
% cd <path_to_autofpga_demo>/sw
% ./wbregs PWRCOUNT
00080008 (PWRCOUNT) : [..o8] 001c6f38
% ./wbregs PWRCOUNT
00080008 (PWRCOUNT) : [.$..] 0024efa5
% ./wbregs PWRCOUNT
00080008 (PWRCOUNT) : [.*.*] 002ad32a
```

PWRCOUNT

Lesson Overview

AutoFPGA Tools

Example AutoFPGA

Running the Demo

▷ PWRCOUNT

RAWREG

In Hardware What just happened?

What just happened?

The Baud Rate

The Buttons

Reconfiguration

GPIO

AutoFPGA

Hardware Emulation

Host Software

The PWRCOUNT register simply counts the number of clock cycles since startup

```
initial r_pwrcount_data = 32'h0;
always @(posedge i_clk)
if (r_pwrcount_data[31])
    r_pwrcount_data[30:0]
        <= r_pwrcount_data[30:0] + 1'b1;
else
    r_pwrcount_data[31:0]
        <= r_pwrcount_data[31:0] + 1'b1;
```

- The top bit saturates, and then stays high
- This logic should look familiar from the last tutorial

We just read this register from within the design

RAWREG

Lesson Overview

AutoFPGA Tools

Example AutoFPGA

Running the Demo

PWRCOUNT

▷ RAWREG

In Hardware

What just happened?

What just happened?

The Baud Rate

The Buttons

Reconfiguration

GPIO

AutoFPGA

Hardware Emulation

Host Software

The design also contains a basic register, RAWREG, that can be read or set

```
% ./wbregs rawreg
0008000c (  RAWREG) : [....] 00000000
% ./wbregs rawreg 0x23458765
0008000c (  RAWREG)-> 23458765
% ./wbregs rawreg
0008000c (  RAWREG) : [.E.e] 23458765
% ./wbregs rawreg 0xdeadbeef
0008000c (  RAWREG)-> deadbeef
% ./wbregs rawreg
0008000c (  RAWREG) : [....] deadbeef
```

We just read our register, set it to `32'h23458765`, read it again, set it to `32'hdeadbeef` and then read it one last time.

RAWREG

Lesson Overview
AutoFPGA Tools
Example AutoFPGA
Running the Demo
PWRCOUNT
▷ RAWREG
In Hardware
What just happened?
What just happened?
The Baud Rate
The Buttons
Reconfiguration
GPIO
AutoFPGA
Hardware Emulation
Host Software

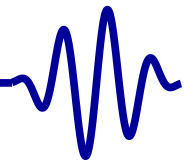
The design also contains a basic register, RAWREG, that can be read or set

```
initial r_rawreg_data = 32'h0;
always @(posedge i_clk)
if (wb_rawreg_stb && wb_rawreg_we)
        r_rawreg_data <= wb_rawreg_data;
```

We just read this register from within the design

- We'll discuss how to build this in more detail in the next section

In Hardware



Lesson Overview

AutoFPGA Tools

Example AutoFPGA

Running the Demo

PWRCOUNT

RAWREG

▷ In Hardware

What just happened?

What just happened?

The Baud Rate

The Buttons

Reconfiguration

GPIO

AutoFPGA

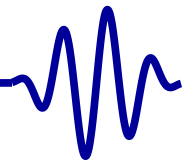
Hardware Emulation

Host Software

To build this design for hardware . . .

- Create a constraint file defining the serial port wires
  - Once done, adjust `auto-data/global.txt`
  - On a Xilinx Vivado design, you'll want to adjust the `@XDC.FILE=` line needs to point to your XDC file
  - On an iCE40 design, you'll want to change that to be a `@PCF.FILE=` line pointing to your PCF file
  - On an ECP5, change it to `@LPF.FILE=`
  - On a Spartan 6, change it to `@UCF.FILE=`
  - In all cases, the file name should follow the = sign

In Hardware



Lesson Overview

AutoFPGA Tools

Example AutoFPGA

Running the Demo

PWRCOUNT

RAWREG

▷ In Hardware

What just happened?

What just happened?

The Baud Rate

The Buttons

Reconfiguration

GPIO

AutoFPGA

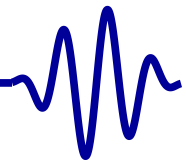
Hardware Emulation

Host Software

To build this design for hardware . . .

- Create a constraint file defining the serial port wires
- Set your incoming clock rate
  - Edit `auto-data/clock.txt`
  - Adjust the `@$CLKFREQHZ=` line to match the clock rate of your design

In Hardware



Lesson Overview

AutoFPGA Tools

Example AutoFPGA

Running the Demo

PWRCOUNT

RAWREG

▷ In Hardware

What just happened?

What just happened?

The Baud Rate

The Buttons

Reconfiguration

GPIO

AutoFPGA

Hardware Emulation

Host Software

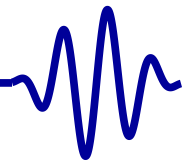
To build this design for hardware . . .

- Create a constraint file defining the serial port wires
- Set your incoming clock rate
- Remove the switch and button configuration
  - These probably won't match your hardware
  - Edit `auto-data/Makefile`
  - Remove the `spio.txt` configuration file from the line beginning with `DATA :=`
  - Rebuild the design from the base project directory

```
% make
```

  - Add the files in the `rtl/` directory to your synthesis flow
  - Build and load the design onto your hardware

In Hardware



Lesson Overview

AutoFPGA Tools

Example AutoFPGA

Running the Demo

PWRCOUNT

RAWREG

▷ In Hardware

What just happened?

What just happened?

The Baud Rate

The Buttons

Reconfiguration

GPIO

AutoFPGA

Hardware Emulation

Host Software

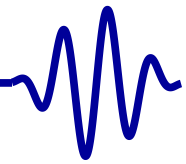
To build this design for hardware . . .

- Create a constraint file defining the serial port wires
- Set your incoming clock rate
- Remove the switch and button configuration
- Open a terminal, and connect netuart to your design

```
% cd <path_to_autofpga_demo>/sw
% netuart /dev/ttyUSBx
```

  - If all works well, you should see a periodic Z printed to the `netuart` terminal
  - This will confirm that you have the right port, and the right serial port settings

In Hardware



Lesson Overview

AutoFPGA Tools

Example AutoFPGA

Running the Demo

PWRCOUNT

RAWREG

▷ In Hardware

What just happened?

What just happened?

The Baud Rate

The Buttons

Reconfiguration

GPIO

AutoFPGA

Hardware Emulation

Host Software

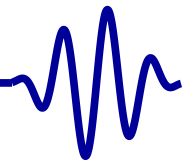
To build this design for hardware . . .

- Create a constraint file defining the serial port wires
- Set your incoming clock rate
- Remove the switch and button configuration
- Open a terminal, and connect netuart to your design

```
% cd <path_to_autofpga_demo>/sw
% netuart /dev/ttyUSBx
```

  - Not all O/S's will support the default 1MBaud data rate
  - If you need to adjust it, change the `@$BAUDRATE=` line in `auto-data/hexbus.txt` to something your O/S will support
  - Then rebuild the project and your hardware design

In Hardware

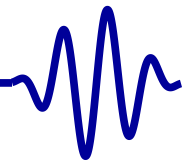


Lesson Overview
AutoFPGA Tools
Example AutoFPGA
Running the Demo
PWRCOUNT
RAWREG
▷ In Hardware
What just happened?
What just happened?
The Baud Rate
The Buttons
Reconfiguration
GPIO
AutoFPGA
Hardware Emulation
Host Software

To build this design for hardware . . .

- Create a constraint file defining the serial port wires
- Set your incoming clock rate
- Remove the switch and button configuration
- Open a terminal, and connect netuart to your design
- Open a second terminal, and run the PWRCOUNT and RAWREG tests as before

What just happened?



Lesson Overview

AutoFPGA Tools

Example AutoFPGA

Running the Demo

PWRCOUNT

RAWREG

In Hardware

What just happened?

What just happened?

The Baud Rate

The Buttons

Reconfiguration

GPIO

AutoFPGA

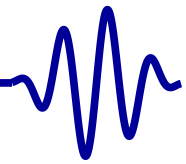
Hardware Emulation

Host Software

What did we just do?

- `make` first builds in the `auto-data/` directory
  - This generates a series of design files, including `main.v` and `toplevel.v`, and other files
- The master `Makefile` then checks if these files have changed
- Changed files are copied from the `auto-generated/` directory into your design
- `make` then built in the `rtl/` directory
  - This called Verilator to convert your Verilog to C++
  - Then built a library from this C++
- `make` then built in the `sim/` directory
  - This built our simulation
  - We could include calls to emulation software here

What just happened?



Lesson Overview

AutoFPGA Tools

Example AutoFPGA

Running the Demo

PWRCOUNT

RAWREG

In Hardware

What just happened?

▷ What just happened?

The Baud Rate

The Buttons

Reconfiguration

GPIO

AutoFPGA

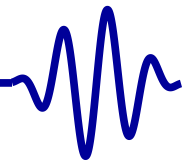
Hardware Emulation

Host Software

What did we just do?

- `make` first builds in the `auto-data/` directory
  - Changed autogenerated files are copied into their respective design locations
- `make` then built in the `rtl/` directory
- `make` then built in the `sim/` directory
- `make` then built in the `sw/` directory
  - This built `wbregs` and `netuart`
  - These depend upon constants and addresses of components (and the baud rate) defined within the design

The Baud Rate



Lesson Overview

AutoFPGA Tools

Example AutoFPGA

Running the Demo

PWRCOUNT

RAWREG

In Hardware

What just happened?

What just happened?

▷ The Baud Rate

The Buttons

Reconfiguration

GPIO

AutoFPGA

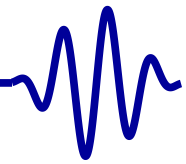
Hardware Emulation

Host Software

How did the baudrate change?

- Changing a baud rate requires changes throughout the design
  - `rtl/main.v`: The RTL design needs to change based upon the design clock frequency
  - `sim/main_tb.cpp`: The simulator needs to know how many clocks per baud at the new rate
  - `sw/netuart.cpp`: The host terminal software needs to know to set to the new baud rate
    - `sw/regdefs.h` contains the new `BAUDRATE` constant
- All this is done by AutoFPGA propagating design constants throughout a series of generated files

The Buttons



Lesson Overview

AutoFPGA Tools

Example AutoFPGA

Running the Demo

PWRCOUNT

RAWREG

In Hardware

What just happened?

What just happened?

The Baud Rate

▷ The Buttons

Reconfiguration

GPIO

AutoFPGA

Hardware Emulation

Host Software

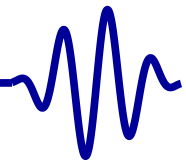
What about removing the buttons and switches?

- The original design referenced 8 LEDs, 5 buttons (with special names), and 8 switches
- By removing one file from the AutoFPGA command line, AutoFPGA ...
  - Removed the processing logic from `rtl/main.v`
  - Removed the `rtl/main.v` ports for these signals
  - Removed the bus ports from the crossbar in `rtl/main.v`
  - Removed the `rtl/toplevel.v` ports for these I/Os
  - Removed the associated host software definitions from `sw/regdefs.h`

AutoFPGA could have also ...

- Removed emulation software from our simulation file
- Left the pin constraints for these ports commented

Reconfiguration



Lesson Overview

AutoFPGA Tools

Example AutoFPGA

Running the Demo

PWRCOUNT

RAWREG

In Hardware

What just happened?

What just happened?

The Baud Rate

The Buttons

▷ Reconfiguration

GPIO

AutoFPGA

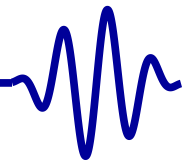
Hardware Emulation

Host Software

AutoFPGA exists for this kind of design automation

- One configuration script per design component
- Removing the configuration script, removes all of the design pieces depending upon that component
- Adding the configuration script puts it back in
- It's more than just adjusting a couple of HDL files
  - C++ sources, headers, and `Makefiles` are adjusted as well
  - The constraint file can also be automatically adjusted as I/Os are added or removed

This is why I like AutoFPGA



Lesson Overview

AutoFPGA Tools

▷ GPIO

GPIO Purpose

Bus Intro

Two Operations

Wishbone Read

Wishbone Write

Select Lines

Caution

Memory Mapping

LED Controller

Button Reader

Connections

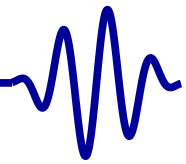
AutoFPGA

Hardware Emulation

Host Software

GPIO

GPIO Purpose



Lesson Overview

AutoFPGA Tools

GPIO

▷ GPIO Purpose

Bus Intro

Two Operations

Wishbone Read

Wishbone Write

Select Lines

Caution

Memory Mapping

LED Controller

Button Reader

Connections

AutoFPGA

Hardware Emulation

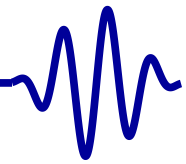
Host Software

Let's now build a GPIO component. Our goals:

- Read external inputs via a bus interface
- Adjust external ouputs, such as LEDs, from the bus

Let's back up a bit first, and discuss how bus reads and writes work

Bus Intro



Lesson Overview

AutoFPGA Tools

GPIO

GPIO Purpose

▷ Bus Intro

Two Operations

Wishbone Read

Wishbone Write

Select Lines

Caution

Memory Mapping

LED Controller

Button Reader

Connections

AutoFPGA

Hardware Emulation

Host Software

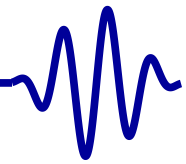
How should software check the value of an input pin?

- Let's check a button, and set an LED if the button is pressed

```
extern  int     _input, _led;

int main(argc, argv) {
    const int ON=1, OFF = 0, BUTTON = 1;

    while(1) {
        if (_input & BUTTON)
            _led = ON;
        else
            _led = OFF;
    }
}
```

- Our goal is to create hardware to implement **_input** and **_led**.

Bus Intro



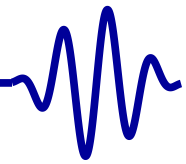
Lesson Overview
AutoFPGA Tools
GPIO
GPIO Purpose
▷ Bus Intro
Two Operations
Wishbone Read
Wishbone Write
Select Lines
Caution
Memory Mapping
LED Controller
Button Reader
Connections
AutoFPGA
Hardware Emulation
Host Software

The host software is similar, but basically the same

- Let's check a button, and set an LED if the button is pressed

```
FPGA    *m_fpga;

int main(argc, argv) {
    FPGAOPEN(m_fpga);
    const int   ON=1, OFF = 0, BUTTON = 1;

    while(1) {
        if (m_fpga->readio(R_INPUT) & BUTTON)
            m_fpga->writeio(R_LED, ON);
        else
            m_fpga->writeio(R_LED, OFF);
    }
}
```

Bus Intro



Lesson Overview

AutoFPGA Tools

GPIO

GPIO Purpose

▷ Bus Intro

Two Operations

Wishbone Read

Wishbone Write

Select Lines

Caution

Memory Mapping

LED Controller

Button Reader

Connections

AutoFPGA

Hardware Emulation

Host Software

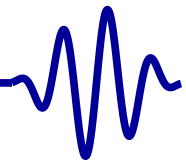
If we can interact with our design from an external host . . .

- Testing gets easier
- Commands can be sent to the FPGA
- Data can be read from or written to the FPGA
  - This includes test data
    Getting test data in and out of an FPGA is a common beginner struggle. We'll start with this capability.

Even before the entire design is in place, . . .

- Can test pieces of it in isolation
- From an external host-based program, or even a shell script

Two Operations



Lesson Overview

AutoFPGA Tools

GPIO

GPIO Purpose

Bus Intro

▷ Two Operations

Wishbone Read

Wishbone Write

Select Lines

Caution

Memory Mapping

LED Controller

Button Reader

Connections

AutoFPGA

Hardware Emulation

Host Software

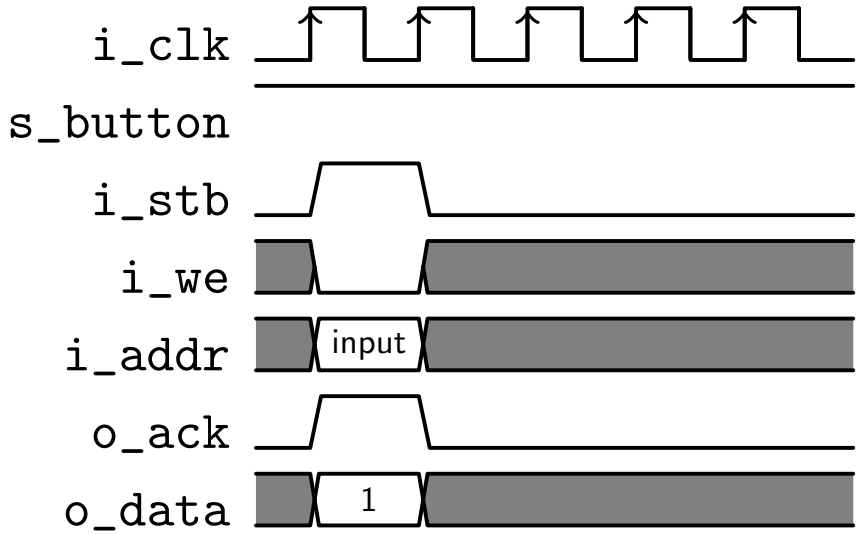
A bus slave needs to support two operations

- Read
  - Given an address
  - Return the value of your (the designers) choice
  - **pressed** = **_input** & **BUTTON**;
    or, from the host,
  - **pressed** = **m_fpga->readio(R_INPUT)** & **BUTTON**;
- Write
  - Given an address and a value
  - Perform an operation of your choice
  - **_led** = (**pressed**) ? **ON**:**OFF**;
    or, from the host,
  - **m_fpga->writeio(LED,** (**pressed**) ? **ON**:**OFF**);

We'll start by implementing the hardware side of this

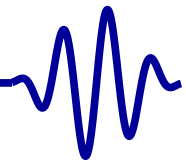
Wishbone Read

Lesson Overview
AutoFPGA Tools
GPIO
GPIO Purpose
Bus Intro
Two Operations
▷ Wishbone Read
Wishbone Write
Select Lines
Caution
Memory Mapping
LED Controller
Button Reader
Connections
AutoFPGA
Hardware Emulation
Host Software



- `STB` requests a bus transaction
- For every `STB && !STALL` one transaction is requested.
  - For now, we'll hold `STALL` low
- If `!WE`, a read is requested
- For each read, `ACK` needs to be set high
- ...with the return data in `DATA`.

Wishbone Read



Lesson Overview
AutoFPGA Tools
GPIO
GPIO Purpose
Bus Intro
Two Operations
Wishbone Read
Wishbone Write
Select Lines
Caution
Memory Mapping
LED Controller
Button Reader
Connections
AutoFPGA
Hardware Emulation
Host Software

The logic to implement this is very straight forward

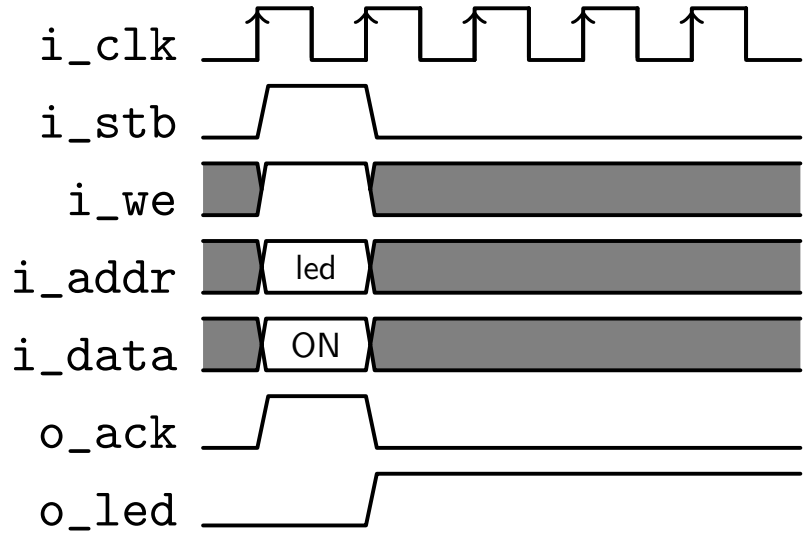
- The slave needs to set `STALL`, `ACK`, and `DATA`

```
assign  o_stall = 0;

// A 2FF synchronizer
always @(posedge i_clk)
        { s_button, button_q }
           <= { button_q, i_button };

// Bus slave read logic
assign  o_ack  = i_stb;
assign  o_data = { 31'h0, s_button };
```

- That's all the slave logic required to read from a button

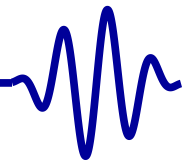
Wishbone Write

Lesson Overview
AutoFPGA Tools
GPIO
GPIO Purpose
Bus Intro
Two Operations
Wishbone Read
▷ Wishbone Write
Select Lines
Caution
Memory Mapping
LED Controller
Button Reader
Connections
AutoFPGA
Hardware Emulation
Host Software



- `STB && WE` requests a write transaction
- For every `STB && !STALL` one transaction is requested.
- If `WE`, a write is requested
- The write data is in the incoming `DATA` wires
- `ACK` gets set once for each request

Wishbone Write



Lesson Overview

AutoFPGA Tools

GPIO

GPIO Purpose

Bus Intro

Two Operations

Wishbone Read

▷ Wishbone Write

Select Lines

Caution

Memory Mapping

LED Controller

Button Reader

Connections

AutoFPGA

Hardware Emulation

Host Software

Let's set our LED from a bus write

- Writes require setting the `STALL` and `ACK` signals

```
assign  o_stall = 1'b0;

always @(posedge i_clk)
if (i_stb && i_we)
        o_led <= i_data[0];

// Acknowledgment is the same as before
assign  o_ack = i_stb;
```

Select Lines

Lesson Overview

AutoFPGA Tools

GPIO

GPIO Purpose

Bus Intro

Two Operations

Wishbone Read

Wishbone Write

▷ Select Lines

Caution

Memory Mapping

LED Controller

Button Reader

Connections

AutoFPGA

Hardware Emulation

Host Software

Wishbone provides byte level access, even if the bus width is wider

- High speed requires accessing many bytes at once
  - The debug bus uses a 32-bit data bus
  - `i_data` and `o_data` are both 32-bits
- CPU's still want to be able to write only 1, or 2, (or more) bytes at a time
  - This is required to implement **char** and **short** data types
- This is accomplished via bus select lines
  - Each bit of `i_sel` indicates which a byte to be written to
  - These may also be called "write strobes" on some busses

Select Lines

Lesson Overview

AutoFPGA Tools

GPIO

GPIO Purpose

Bus Intro

Two Operations

Wishbone Read

Wishbone Write

▷ Select Lines

Caution

Memory Mapping

LED Controller

Button Reader

Connections

AutoFPGA

Hardware Emulation

Host Software

Using bus select lines to write to a memory

```
parameter DW = 32; // The bus data width
reg      [DW-1:0] mem [MEMSZ-1:0];

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
```

Select Lines

Lesson Overview

AutoFPGA Tools

GPIO

GPIO Purpose

Bus Intro

Two Operations

Wishbone Read

Wishbone Write

▷ Select Lines

Caution

Memory Mapping

LED Controller

Button Reader

Connections

AutoFPGA

Hardware Emulation

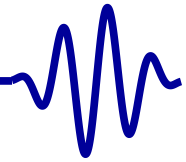
Host Software

Using a **for** loop, we can make this logic generic across all bus data widths, `DW`

```
integer k;

// Write to memory upon request
always @(posedge i_clk)
if (i_stb && i_we)
begin
    for(k=0; k<(DW/8); k=k+1)
    if (i_sel[k])
        mem[i_addr][8*k +: 8] <= i_data[8*k +: 8];
end
```

You now know how to make a basic Wishbone peripheral

Caution



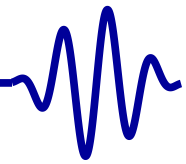
Lesson Overview
AutoFPGA Tools
GPIO
GPIO Purpose
Bus Intro
Two Operations
Wishbone Read
Wishbone Write
Select Lines
Caution
Memory Mapping
LED Controller
Button Reader
Connections
AutoFPGA
Hardware Emulation
Host Software

Be careful with **for** loops

- They don't have the same meaning they do in software
- In HDL languages, **for** loops generate hardware
  - They are *always* unrolled
  - Each loop iteration generates another copy of (nearly) the same hardware
- Loops depending on prior results typically don't do what you want
- This loop creates multiple copies of the same circuit
  - All references within the loop are to different inputs
  - All values set are different outputs

Always try to be aware of the logic you are generating

Select Lines



Lesson Overview

AutoFPGA Tools

GPIO

GPIO Purpose

Bus Intro

Two Operations

Wishbone Read

Wishbone Write

Select Lines

Caution

Memory Mapping

LED Controller

Button Reader

Connections

AutoFPGA

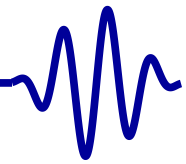
Hardware Emulation

Host Software

The debugging bus sets all of the select lines

- It's important to know how these work, but . . .
- We won't be using them for a while

Memory Mapping



Lesson Overview
AutoFPGA Tools
GPIO
GPIO Purpose
Bus Intro
Two Operations
Wishbone Read
Wishbone Write
Select Lines
Caution
Memory Mapping
LED Controller
Button Reader
Connections
AutoFPGA
Hardware Emulation
Host Software

Using a bus interface is very versatile

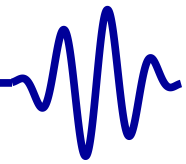
- It can be used for things other than memory

Example: Buttons and LEDs

- LED
  Writes to the LED slave change LED states
- Buttons
  Reads from a button register might read the state of the buttons: pressed or not

This follows from our example above. Are there other useful examples?

Memory Mapping



Lesson Overview

AutoFPGA Tools

GPIO

GPIO Purpose

Bus Intro

Two Operations

Wishbone Read

Wishbone Write

Select Lines

Caution

Memory Mapping

LED Controller

Button Reader

Connections

AutoFPGA

Hardware Emulation

Host Software

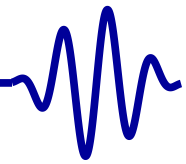
Using a bus interface is very versatile

- It can be used for things other than memory

Example: Serial port

- Transmit address
  Writes to this address send characters
- Receive address
  Reads from this address receive characters or a "no-character is available" signal

Memory Mapping



Lesson Overview
AutoFPGA Tools
GPIO
GPIO Purpose
Bus Intro
Two Operations
Wishbone Read
Wishbone Write
Select Lines
Caution
Memory Mapping
LED Controller
Button Reader
Connections
AutoFPGA
Hardware Emulation
Host Software

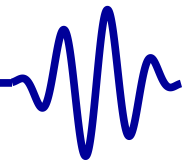
Using a bus interface is very versatile

- It can be used for things other than memory

Example: Serial port with FIFO

- Transmit address
  Writes to this address enqueue characters for transmission
- Receive address
  Reads from this address dequeue characters received or a "no-character is available" signal
- Control Register
  Sets the baud rate, # of data bits, type/kind of parity, # of stop bits
- Status Register
  Returns how full the receive FIFO is
  Returns how empty the transmit FIFO is

Memory Mapping



Lesson Overview
AutoFPGA Tools
GPIO
GPIO Purpose
Bus Intro
Two Operations
Wishbone Read
Wishbone Write
Select Lines
Caution
Memory Mapping
LED Controller
Button Reader
Connections
AutoFPGA
Hardware Emulation
Host Software

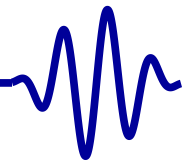
Using a bus interface is very versatile

- It can be used for things other than memory

Example: Video

- Frame buffer
  An area of memory that is read directly to the screen, but can also be read or written from any other bus master–such as an embedded CPU.

LED Controller



Lesson Overview

AutoFPGA Tools

GPIO

GPIO Purpose

Bus Intro

Two Operations

Wishbone Read

Wishbone Write

Select Lines

Caution

Memory Mapping

▷ LED Controller

Button Reader

Connections

AutoFPGA

Hardware Emulation

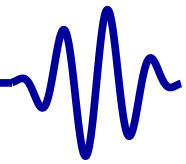
Host Software

Let's finish off our button and LED controllers

- By placing them into modules

```
module ledcontrol(i_clk, i_reset,
        i_wb_cyc, i_wb_stb, i_wb_we, // i_wb_addr,
        i_wb_data, i_wb_sel,
        o_wb_stall, o_wb_ack, o_wb_data,
        o_led);
        // Declarations ...
        input wire i_clk, i_reset;
        input wire i_wb_cyc, i_wb_stb, i_wb_we;
        input wire [DW-1:0] i_wb_data;
        input wire [DW/8-1:0] i_wb_sel;
        //
        output wire o_wb_stall, o_wb_ack;
        output wire [DW-1:0] o_wb_data;
        output reg  o_led;
```

LED Controller



Lesson Overview

AutoFPGA Tools

GPIO

GPIO Purpose

Bus Intro

Two Operations

Wishbone Read

Wishbone Write

Select Lines

Caution

Memory Mapping

▷ LED Controller

Button Reader

Connections

AutoFPGA

Hardware Emulation

Host Software

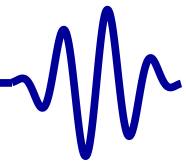
Let's finish off our button and LED controllers

- By placing them into modules

```
module ledcontrol(// ...

        // Bus slave logic
        assign  o_wb_stall = 1'b0;
        assign  o_wb_ack   = i_wb_stb;
        assign  o_wb_data  = { 31'h0, o_led };

        always @(posedge i_clk)
        if (i_wb_stb && i_wb_we)
                o_led <= i_wb_data[0];
endmodule
```

AutoFPGA calls a peripheral of this type a SINGLE peripheral

Button Reader



Lesson Overview
AutoFPGA Tools
GPIO
GPIO Purpose
Bus Intro
Two Operations
Wishbone Read
Wishbone Write
Select Lines
Caution
Memory Mapping
LED Controller
▷ Button Reader
Connections
AutoFPGA
Hardware Emulation
Host Software

Let's finish off our button and LED controllers

- By placing them into modules

```
module buttonreader(// ...

        // Bus slave logic
        assign  o_wb_stall = 1'b0;
        assign  o_wb_ack   = i_wb_stb;
        assign  o_wb_data  = { 31'h0, s_button };

        // Don't forget the 2FF synchronizer
        always @(posedge i_clk)
                { s_button, q_button }
                 <= { q_button, i_button };
endmodule
```

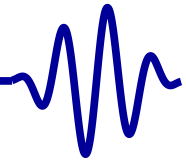
AutoFPGA would also call this a SINGLE peripheral

Connections

Lesson Overview
AutoFPGA Tools
GPIO
GPIO Purpose
Bus Intro
Two Operations
Wishbone Read
Wishbone Write
Select Lines
Caution
Memory Mapping
LED Controller
Button Reader
▷ Connections
AutoFPGA
Hardware Emulation
Host Software

The last step will be to connect these peripherals to our design

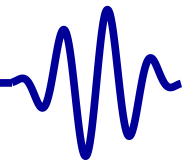
- We could do this manually
  - Need to pick an address for each peripheral
  - Possibly adjust other peripheral addresses to make room
  - Write address decoding logic
  - Connect this to some form of bus interconnect
  - Adjust C/C++ pointers and constants referencing these addresses
  - Connect any hardware emulators
- This is all a lot of busy work
  - It's easy to do, annoying to do again and again
  - AutoFPGA will help us here



Lesson Overview
AutoFPGA Tools
GPIO
▷ AutoFPGA
Config Files
The Clock
The Bus
Bus Master
LED Slave
Makefile
Type SINGLE
Your turn
Hardware Emulation
Host Software

AutoFPGA

Config Files

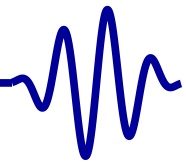


Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
▷ Config Files
The Clock
The Bus
Bus Master
LED Slave
Makefile
Type SINGLE
Your turn
Hardware Emulation
Host Software

AutoFPGA composes a design from components

- Each component has a configuration file
- This file contains a series of `@TAG=VALUE` lines
- `VALUE` is typically just a piece of text
  - It may consume less than a line
  - It may take up many lines
- The `VALUE` may then be referenced later using `@$(TAG)`
- `@$TAG=VALUE` with a dollar sign sets a numeric value
  - AutoFPGA provides some ability for expression evaluation
  - But only if the tag is defined using `@$`
- The first key, `@PREFIX`, is required
  - This gives the component its name
  - All following keys now belong to this component

Config Files



Lesson Overview

AutoFPGA Tools

GPIO

AutoFPGA

▷ Config Files

The Clock

The Bus

Bus Master

LED Slave

Makefile

Type SINGLE

Your turn

Hardware Emulation

Host Software

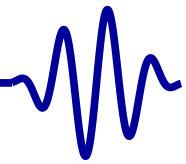
AutoFPGA composes a design from components

- Each component has a configuration file
- Key names are hierarchical
- For example, to define a bus we define a set of bus tags
  - `@BUS.NAME=wb`
  - `@BUS.TYPE=wb`
  - `@BUS.CLOCK=clk`
- Two types of comment lines
  - Comments either begin with ##, or
  - With a # and a space.
  - Anything else is aggregated into the previous value

The goal is to provide a copy/paste utility

- With an ability for cross file variable/value expansion

The Clock



Lesson Overview

AutoFPGA Tools

GPIO

AutoFPGA

Config Files

▷ The Clock

The Bus

Bus Master

LED Slave

Makefile

Type SINGLE

Your turn

Hardware Emulation

Host Software

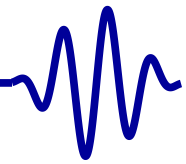
The first step is to define a clock

- Looking back at the `autofpga-demo`
- The clock is defined in `auto-data/clock.txt`

```
@$CLKFREQHZ=100000000    An AutoFPGA global variable
@PREFIX=clk              ...
```

- Values defined before `@PREFIX=` have global scope
  - They be referenced anywhere else in the design
  - Example: `@$X=@$(CLKFREQHZ)`
- Values defined after would need to be referenced by the prefix first
  - Example: `@$X=@$(clk.CLOCK.FREQUENCY)`

The Clock



Lesson Overview

AutoFPGA Tools

GPIO

AutoFPGA

Config Files

▷ The Clock

The Bus

Bus Master

LED Slave

Makefile

Type SINGLE

Your turn

Hardware Emulation

Host Software

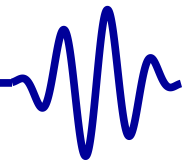
The first step is to define a clock

- Looking back at the `autofpga-demo`
- The clock is defined in `auto-data/clock.txt`

```
@$CLKFREQHZ=100000000
@PREFIX=clk              Gives this component a name
@CLOCK.NAME=clk          We'll call this clock, clk
@CLOCK.TOP=i_clk         Name of the clock input wire
```

- Let's tell AutoFPGA about a clock we'll just call `clk`
- By defining a `CLOCK.TOP` tag, this value will be included in the port list of `toplevel.v`
- This is unique to clocks
  - Other ports are defined differently

The Clock



Lesson Overview

AutoFPGA Tools

GPIO

AutoFPGA

Config Files

▷ The Clock

The Bus

Bus Master

LED Slave

Makefile

Type SINGLE

Your turn

Hardware Emulation

Host Software

The first step is to define a clock

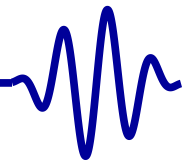
- Looking back at the `autofpga-demo`
- The clock is defined in `auto-data/clock.txt`

`@$CLKFREQHZ=100000000`	*An AutoFPGA global variable*
`@PREFIX=clk`	*Gives this component a name*
`@CLOCK.TOP=i_clk`	*Name of the clock input wire*
`@CLOCK.NAME=clk`	*We'll call this clk*

- Finally, let's tell AutoFPGA the frequency of this clock

  `@CLOCK.FREQUENCY=@$(CLKFREQHZ)`
- Note how we just referenced the global value defined above

The Clock



Lesson Overview

AutoFPGA Tools

GPIO

AutoFPGA

Config Files

▷ The Clock

The Bus

Bus Master

LED Slave

Makefile

Type SINGLE

Your turn

Hardware Emulation

Host Software

Our clock needs a minimum of logic at the top level

- Still looking in `auto-data/clock.txt`
- The `@TOP.DEFNS` value gets pasted into the top of `toplevel.v`

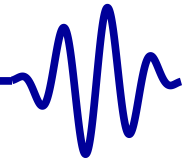
```
@TOP.DEFNS=
                wire s_clk, s_reset;
```

- The `@TOP.INSERT` value gets pasted in further down

```
@TOP.INSERT=
                assign s_reset = 1'b0;
                assign s_clk = i_clk;
```

- Since these are copy/paste lines, they can contain any logic you deem fit
- AutoFPGA requires definitions for the two special signals, `s_clk` and `s_reset`.

The Bus



Lesson Overview

AutoFPGA Tools

GPIO

AutoFPGA

Config Files

The Clock

▷ The Bus

Bus Master

LED Slave

Makefile

Type SINGLE

Your turn

Hardware Emulation

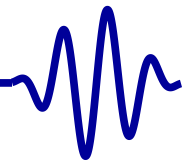
Host Software

The next step is to define a bus

- Looking back at the `autofpga-demo`
- The bus is defined in `auto-data/global.txt`

`@BUS.NAME=wb`	*Define a bus named wb*
`@BUS.TYPE=wb`	*It's a Wishbone bus*
`@BUS.WIDTH=32`	*With a 32-bit data width*
`@BUS.CLOCK=clk`	*Using the clock named clk*

Bus Master



Lesson Overview

AutoFPGA Tools

GPIO

AutoFPGA

Config Files

The Clock

The Bus

▷ Bus Master

LED Slave

Makefile

Type SINGLE

Your turn

Hardware Emulation

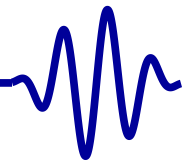
Host Software

To be useful, every bus needs a bus master

- The debugging bus master is defined in `auto-data/hexbus.txt`
  This may look complex at first glance. It's not really that bad. Over the course of this tutorial, we'll work through most of what these tags mean–so you can generate and connect your own bus masters.
- This particular version drives a Wishbone bus
  - Another debug bus could be created to drive an AXI bus
  - Alternatively, a bridge could convert from Wishbone to AXI

You can learn more of how this particular debugging bus works from the blog.

LED Slave



Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Config Files
The Clock
The Bus
Bus Master
▷ LED Slave
Makefile
Type SINGLE
Your turn
Hardware Emulation
Host Software

We can now define our LED slave

`@PREFIX=led`	*Define a slave named led*
`@NADDR=1`	*Containing one word*
`@SLAVE.TYPE=SINGLE`	*More on this later*
`@SLAVE.BUS=wb`	*Connect to bus named wb*
`@MAIN.PORTLIST=`	*Define a design port*

```
        o_led
```

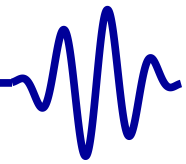
`@MAIN.IODECL=` *Declare our output*

```
    output wire o_led;
```

`@MAIN.INSERT=` *Our main logic*

```
    ledcontrol
    theled(i_clk,
        @$(SLAVE.PORTLIST), // AutoConnect
        o_led);
```

LED Slave



Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Config Files
The Clock
The Bus
Bus Master
▷ LED Slave
Makefile
Type SINGLE
Your turn
Hardware Emulation
Host Software

There's one last step to our definition

- We need to define a register we can read from

```
@REGS.N=1                    Define a single register
@REGS.0=0 R_LED LED
```

This works for the host interface

- It defines a C++ constant **R_LED**
- Having the value of our LED control register's address
- It also defines a human readable name, "LED"
- We can now use this with wbregs
- The 0 value just specifies the offset of our register (in words) from the LED controller's base address

```
% ./wbregs led
```

Makefile

Lesson Overview

AutoFPGA Tools

GPIO

AutoFPGA

Config Files

The Clock

The Bus

Bus Master

LED Slave

▷ Makefile

Type SINGLE

Your turn

Hardware Emulation

Host Software

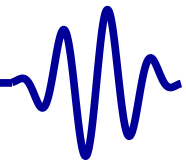
Now that we've created a new component, you'll now need to adjust the `auto-data/Makefile`

- This is as simple as adding the name of our new config file to the `DATA :=` definition line
- Now re-run `make` and check out the differences

You should see differences in ...

- `rtl/toplevel.v`
- `rtl/main.v`
- `sw/regdefs.h`, and
- `sw/regdefs.cpp`

Type SINGLE

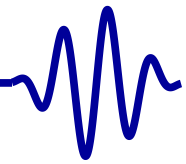


Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Config Files
The Clock
The Bus
Bus Master
LED Slave
Makefile
Type SINGLE
Your turn
Hardware Emulation
Host Software

AutoFPGA supports four special classes of bus slaves

- `SINGLE` (this example)
  - Describes a peripheral having only a single address
  - The slave is not allowed to stall the bus
  - Allows AutoFPGA to simplify the bus logic
  - For Wishbone, AutoFPGA ignores the `STALL` and `ACK` signals of SINGLE peripherals
    - `STALL` is assumed to be zero
    - `ACK` is assumed to be equal to `STB`

Type SINGLE

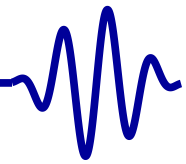


Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Config Files
The Clock
The Bus
Bus Master
LED Slave
Makefile
▷ Type SINGLE
Your turn
Hardware Emulation
Host Software

AutoFPGA supports four special classes of bus slaves

- `SINGLE` (this example)
  - Describes a peripheral having only a single address
  - The slave is not allowed to stall the bus
  - Allows AutoFPGA to simplify the bus logic
  - For Wishbone, AutoFPGA ignores the `STALL` and `ACK` signals of SINGLE peripherals
  - For AXI or AXI-Lite, AutoFPGA sets all the `xREADY` lines high, and ignores (assumes) `BVALID` and `RVALID`
    - `xREADY` is assumed to be one
    - `AWVALID` is guaranteed to be equal to `WVALID`
    - `xVALID` is assumed to true one clock after `AxVALID`
    - A slave driver handles the conversion to the simpler bus standard

Type SINGLE



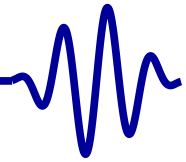
Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Config Files
The Clock
The Bus
Bus Master
LED Slave
Makefile
▷ Type SINGLE
Your turn
Hardware Emulation
Host Software

AutoFPGA supports four special classes of bus slaves

- `SINGLE` (this example)
- `DOUBLE`
  - Consumes one clock to generate the result
  - Allows the slave to select a register from among multiple possible return addresses
- `MEMORY`
  - A generic bus slave, but one needing a linker script entry
  - Bus logic in this case is identical to `OTHER` below
- `OTHER`
  - Everything else

Later examples in this course will explore other types of bus slaves

Your turn



Lesson Overview

AutoFPGA Tools

GPIO

AutoFPGA

Config Files

The Clock

The Bus

Bus Master

LED Slave

Makefile

Type SINGLE

Your turn

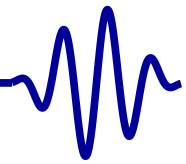
Hardware Emulation

Host Software

We just built `ledcontrol`, you finish `buttonreader.v`

- Create the Verilog file
- Create a (nearly identical) AutoFPGA configuration
- Match the portlist to what AutoFPGA gives you

Your turn

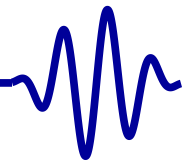


Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Config Files
The Clock
The Bus
Bus Master
LED Slave
Makefile
Type SINGLE
Your turn
Hardware Emulation
Host Software

You should now be able to test your design with `wbregs`

- `wbregs led 1` # *should turn your LED on*
- `wbregs led 0` # *should turn your LED off*
- `wbregs input` # *should read your button status register*

All we need now is some host software to adjust the LED automatically



Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Hardware Emulation
Emulator
Host Software

Hardware Emulation

Emulator

Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Hardware Emulation
Emulator
Host Software

We wrote a button emulator in lesson 7 of the beginners tutorial

- We can use that same emulator again now
- Let's now add a call to that emulator to `buttonreader.txt`
- We'll first declare it

```
@SIM.CLOCK=clk
@SIM.INCLUDE=
```

```
#include "buttonsim.h"
```

Emulator

Lesson Overview

AutoFPGA Tools

GPIO

AutoFPGA

Hardware Emulation

▷ Emulator

Host Software

We wrote a button emulator in lesson 7 of the beginners tutorial

- We can use that same emulator again now
- Let's now add a call to that emulator to `buttonreader.txt`
- We'll first declare it
- Then define it

```
@SIM.CLOCK=clk
@SIM.INCLUDE=    // ...
@SIM.DEFNS=
    BUTTONSIM       *m_button;
```

Emulator

Lesson Overview

AutoFPGA Tools

GPIO

AutoFPGA

Hardware Emulation

▷ Emulator

Host Software

We wrote a button emulator in lesson 7 of the beginners tutorial

- We can use that same emulator again now
- Let's now add a call to that emulator to `buttonreader.txt`
- We'll first declare it
- Then define it
- Then initialize it

```
@SIM.CLOCK=clk
@SIM.INCLUDE=    // ...
@SIM.DEFNS=      // ...
@SIM.INIT=
```

```
    m_button = new BUTTONSIM();
```

Emulator

Lesson Overview

AutoFPGA Tools

GPIO

AutoFPGA

Hardware Emulation

▷ Emulator

Host Software

We wrote a button emulator in lesson 7 of the beginners tutorial

- We can use that same emulator again now
- Let's now add a call to that emulator to `buttonreader.txt`
- We'll first declare it
- Then define it
- Then initialize it
- Then call the emulator on every clock tick

```
@SIM.CLOCK=clk
@SIM.INCLUDE=    // ...
@SIM.DEFNS=      // ...
@SIM.INIT=       // ...
@SIM.TICK=
```

```
m_core->i_button = (*m_button)();
```

Emulator

Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Hardware Emulation
Emulator
Host Software

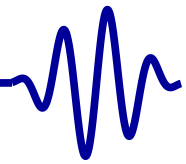
We wrote a button emulator in lesson 7 of the beginners tutorial

- We can use that same emulator again now
- Let's now add a call to that emulator to `buttonreader.txt`
- We'll first declare it
- Then define it
- Then initialize it
- Then call the emulator on every clock tick

These lines will just get copied and pasted into your `sim/main_tb.cpp` file

- When/if you remove `buttonreader.txt` from your config,
- They'll then be removed as well

LED Emulation



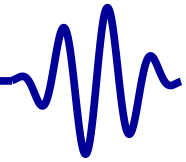
Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Hardware Emulation
Emulator
Host Software

Emulating the LED is easier

- You can just declare an **m_lastled** variable
- Set it on every clock to **m_core->o_led**
- Print every time it changes
- This will print to the `main_tb` console
  - Together with anything else
  - The debug bus will print to the console as well

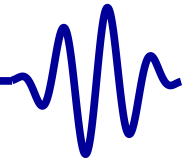
Go ahead, try it.

Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Hardware Emulation
Host Software
Software
Building
Embedded CPU
Your turn



Host Software

Software



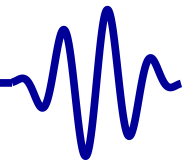
Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Hardware Emulation
Host Software
▷ Software
Building
Embedded CPU
Your turn

Let's now build the blinking software we described earlier

```
#include "port.h"     // Def'n FPGAOPEN
#include "regdefs.h" // Def'n registers
#include "hexbus.h"  // Def'n the FPGA interface

FPGA *m_fpga;    // Declare an FPGA interface
int main(int argc, char **argv) {
        // Connect to our FPGA
        FPGAOPEN(m_fpga);

        // The rest follows from before
        // ...
        //
```

Save this as `blinker.cpp`

Building



Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Hardware Emulation
Host Software
Software
▷ Building
Embedded CPU
Your turn

To build this, we'll adjust the `sw/Makefile`

- Add `blinker` to the list of `PROGRAMS`
- Add the following lines

```
blinker: $(OBJDIR)/blinker.o $(BUSOBJS)
        $(CXX) $(CFLAGS) $^ $(LIBS) -o $@
```

Running `make` should now build your blinking program

Embedded CPU

Lesson Overview

AutoFPGA Tools

GPIO

AutoFPGA

Hardware Emulation

Host Software

Software

Building

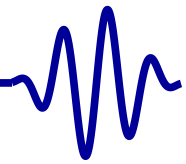
▷ Embedded CPU

Your turn

The software for an embedded CPU would be quite similar

- Later on, we'll add in the ZipCPU
- Until then, we'll hold off developing embedded CPU software
- This will allow us to focus on building and debugging the environment the CPU will eventually be placed within

Your turn



Lesson Overview
AutoFPGA Tools
GPIO
AutoFPGA
Hardware Emulation
Host Software
Software
Building
Embedded CPU
▷ Your turn

Your turn:

- Try out `blinker`
- Does the LED turn on when you press the button?
- Does the LED turn off when you release the button?

What if you modify the host program?

- Can you make the LED blink by running a program on your PC?