

# UP\_WISHBONE\_CLASSIC



November 18, 2024

Jay Convertino

# Contents

<b>1 Usage</b>	<b>2</b>
1.1 Introduction . . . . .	2
1.2 Dependencies . . . . .	2
1.2.1 fusesoc_info Depenecies . . . . .	2
1.3 In a Project . . . . .	2
<b>2 Architecture</b>	<b>5</b>
<b>3 Building</b>	<b>6</b>
3.1 fusesoc . . . . .	6
3.2 Source Files . . . . .	6
3.2.1 fusesoc_info File List . . . . .	6
3.3 Targets . . . . .	6
3.3.1 fusesoc_info Targets . . . . .	6
3.4 Directory Guide . . . . .	7
<b>4 Simulation</b>	<b>8</b>
4.1 iverilog . . . . .	8
4.2 cocotb . . . . .	8
<b>5 Module Documentation</b>	<b>9</b>
5.1 up_wishbone_classic . . . . .	10

# 1 Usage

## 1.1 Introduction

This core converts the Wishbone Classic bus to the uP bus. This allows any core with a uP bus to be interfaced with a Wishbone Classic bus. Combination of combinatorial logic and synchronous logic.

## 1.2 Dependencies

The following are the dependencies of the cores.

- fusesoc 2.X
- iverilog (simulation)
- cocotb (simulation)

### 1.2.1 fusesoc\_info Dependencies

- dep
  - AFRL:utility:helper:1.0.0
  - ::wb\_common:1.0.3

## 1.3 In a Project

This core is made to interface Wishbone Classic bus to uP based device cores. This is part of a family of converters based on Analog Devices uP specification. Using this allows usage of Analog Devices AXI Lite core, AFRL APB3, AFRL Wishbone Classic, and AFRL Wishbone Pipeline converters. Meaning any uP core can be easily customized to any bus quickly. These are made for relatively slow speed bus device interfaces. An example of a Verilog uP interface provided below.

```
//output signals assigned to registers.
assign up_rack  = r_up_rack & up_rreq;
assign up_wack  = r_up_wack & up_wreq;
assign up_rdata = r_up_rdata;
assign irq      = r_irq;

assign s_rx_ren = ((up_raddr[3:0] == RX_FIFO_REG) &&
    ↪ up_rreq ? r_up_rack & r_rx_ren : 0);

//up registers decoder
always @(posedge clk)
```

```

begin
  if(rstn == 1'b0)
    begin
      r_up_rack    <= 1'b0;
      r_up_wack    <= 1'b0;
      r_up_rdata   <= 0;

      r_rx_ren     <= 1'b0;

      r_overflow   <= 1'b0;

      r_control_reg <= 0;
    end else begin
      r_up_rack    <= 1'b0;
      r_up_wack    <= 1'b0;
      r_tx_wen     <= 1'b0;
      r_rx_ren     <= 1'b0;
      r_up_rdata   <= r_up_rdata;
      //clear reset bits
      r_control_reg[RESET_RX_BIT] <= 1'b0;
      r_control_reg[RESET_TX_BIT] <= 1'b0;

      if(rx_full == 1'b1)
        begin
          r_overflow <= 1'b1;
        end

      //read request
      if(up_rreq == 1'b1)
        begin
          r_up_rack <= 1'b1;

          case(up_raddr[3:0])
            RX_FIFO_REG: begin
              r_up_rdata <= rx_rdata & {{(BUS_WIDTH*8-
                ↪ DATA_BITS){1'b0}}, {DATA_BITS{1'b1}}}};
              r_rx_ren <= 1'b1;
            end
            STATUS_REG: begin
              r_up_rdata <= {{(BUS_WIDTH*8-8){1'b0}},
                ↪ s_parity_err, s_frame_err, r_overflow,
                ↪ r_irq_en, tx_full, tx_empty, rx_full,
                ↪ rx_valid};
              r_overflow <= 1'b0;
            end
            default: begin

```

```

        r_up_rdata <= 0;
    end
endcase
end

//write request
if(up_wreq == 1'b1)
begin
    r_up_wack <= 1'b1;

    //only allow write once ack (Analog Devices does
    ↪ the same)
    if(r_up_wack == 1'b1) begin
        case(up_waddr[3:0])
            TX_FIFO_REG: begin
                r_tx_wdata <= up_wdata;
                r_tx_wen <= 1'b1;
            end
            CONTROL_REG: begin
                r_control_reg <= up_wdata;
            end
            default: begin
            end
        endcase
    end
end
end

//up control register processing and fifo reset
always @(posedge clk)
begin
    if(rstn == 1'b0)
    begin
        r_rstn_rx_delay <= ~0;
        r_rstn_tx_delay <= ~0;
        r_irq_en <= 1'b0;
    end else begin
        r_rstn_rx_delay <= {1'b1, r_rstn_rx_delay[
            ↪ FIFO_DEPTH-1:1]};
        r_rstn_tx_delay <= {1'b1, r_rstn_tx_delay[
            ↪ FIFO_DEPTH-1:1]};

        if(r_control_reg[RESET_RX_BIT])
        begin
            r_rstn_rx_delay <= {FIFO_DEPTH{1'b0}};
        end
    end
end

```

```

    end

    if(r_control_reg[RESET_TX_BIT])
    begin
        r_rstn_tx_delay <= {FIFO_DEPTH{1'b0}};
    end

    if(r_control_reg[ENABLE_INTR_BIT] != r_irq_en)
        r_irq_en <= r_control_reg[ENABLE_INTR_BIT];
    end
end
end

```

## 2 Architecture

The only module is the up\_wishbone\_classic module. It is listed below.

- **up\_wishbone\_classic** Convert Wishbone Classic to the Analog Devices uP BUS. (see core for documentation).

This core has two generate blocks, and two always blocks. They are listed below.

generate:

1. Part Select Write generates uP signals that have the non-selected elements blanked out with zeros.
2. Part Select Read generates Wishbone classic signals that have non-selected elements blanked out with zeros.

always:

- Burst State Control
  1. Based on Wishbone Classic CTI state, change request logic.
  2. If CTI burst does not match a configuration, no request is allowed.
  3. Check the address state, if we are currently requesting burst mode then change to self incrementing address.
  4. If we hit the last in burst mode, switch out of it.
- Reset Hold, holds reset for 8 more clock cycles to comply with Wishbone Standard.

Please see 5 for more information.

## 3 Building

The Wishbone Classic core is written in Verilog 2001. They should synthesize in any modern FPGA software. The core comes as a fusesoc packaged core and can be included in any other core. Be sure to make sure you have met the dependencies listed in the previous section.

### 3.1 fusesoc

Fusesoc is a system for building FPGA software without relying on the internal project management of the tool. Avoiding vendor lock in to Vivado or Quartus. These cores, when included in a project, can be easily integrated and targets created based upon the end developer needs. The core by itself is not a part of a system and should be integrated into a fusesoc based system. Simulations are setup to use fusesoc and are a part of its targets.

### 3.2 Source Files

#### 3.2.1 fusesoc\_info File List

- src
  - Type: verilogSource
  - src/up\_wishbone\_classic.v
- tb
  - Type: verilogSource
  - tb/tb\_wishbone\_slave.v

### 3.3 Targets

#### 3.3.1 fusesoc\_info Targets

- default
  - Info: Default for IP integration.
  - src
  - dep
- sim
  - Info: Base simulation using icarus as default.
  - src
  - dep
  - tb

### 3.4 Directory Guide

Below highlights important folders from the root of the directory.

1. **docs** Contains all documentation related to this project.
  - **manual** Contains user manual and github page that are generated from the latex sources.
  - **specs** Contains specifications for the bus.
2. **src** Contains source files for the core
3. **tb** Contains test bench files for iverilog and cocotb
  - **cocotb** testbench files



## **4 Simulation**

There are a few different simulations that can be run for this core.

### **4.1 iverilog**

iverilog is used for simple test benches for quick verification, visually, of the core.

### **4.2 cocotb**

Future simulations will use cocotb. This feature is not yet implemented.

## 5 Module Documentation

There is a single async module for this core.

- **up\_wishbone\_classic** Wishbone Classic to uP converter

The next sections document the module in great detail.

# up\_wishbone\_classic.v

---

## AUTHORS

---

JAY CONVERTINO

---

## DATES

---

2024/03/01

---

## INFORMATION

---

### Brief

---

Wishbone Classic slave to uP interface

### License MIT

---

Copyright 2024 Jay Convertino

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## up\_wishbone\_classic

---

```
module up_wishbone_classic #(
    parameter
    ADDRESS_WIDTH
    =
    16,
    parameter
    BUS_WIDTH
    =
    4
) ( input clk, input rst, input s_wb_cyc, input s_wb_stb, input s_wb_we, in
```

Wishbone Classic slave to uP up\_wishbone\_classic

## Parameters

**ADDRESS\_WIDTH** Width of the Wishbone address port in bits.

parameter

**BUS\_WIDTH** Width of the Wishbone bus data port in bytes.

parameter

## Ports

<b>clk</b>	Clock
<b>rst</b>	Positive reset
<b>s_wb_cyc</b>	Bus Cycle in process
<b>s_wb_stb</b>	Valid data transfer cycle
<b>s_wb_we</b>	Active High write, low read
<b>s_wb_addr</b>	Bus address
<b>s_wb_data_i</b>	Input data
<b>s_wb_sel</b>	Device Select
<b>s_wb_bte</b>	Burst Type Extension
<b>s_wb_cti</b>	Cycle Type
<b>s_wb_ack</b>	Bus transaction terminated
<b>s_wb_data_o</b>	Output data
<b>s_wb_err</b>	Active high when a bus error is present
<b>up_rreq</b>	uP bus read request
<b>up_rack</b>	uP bus read ack
<b>up_raddr</b>	uP bus read address
<b>up_rdata</b>	uP bus read data
<b>up_wreq</b>	uP bus write request
<b>up_wack</b>	uP bus write ack
<b>up_waddr</b>	uP bus write address
<b>up_wdata</b>	uP bus write data

## VARIABLES

---

### s\_next\_address

---

```
assign s_next_address = wb_next_adr(
    r_address,
    r_wb_cti &
    s_wb_cti,
    s_wb_bte,
    BUS_WIDTH *
    8
)
```

Use the fusesoc wb\_next\_adr function to generate a address when wishbone classic is in a burst mode.

## valid

---

```
assign valid = s_wb_cyc & s_wb_stb & ~r_rst[0]
```

Indicate valid request from wishbone.

## up\_rreq

---

```
assign up_rreq = ~s_wb_we & r_req
```

Convert wishbone read requests to up requests

## up\_wreq

---

```
assign up_wreq = s_wb_we & r_req
```

Convert wishbone write requests to up requests

## s\_wb\_err

---

```
assign s_wb_err = 1'b0
```

TODO:check for burst address errors

## up\_raddr

---

```
assign up_raddr = (
    address_state == init_address ? s_wb_addr : s_next_address)
    :
    0
    )
    ~s_wb_we & ~r_rst[0] ? (
```

assign address to read address port if selected

## up\_waddr

---

```
assign up_waddr = (
    address_state == init_address ? s_wb_addr : r_address)
    :
    0
    )
    s_wb_we & ~r_rst[0] ? (
```

assign address to write address port if selected

## up\_ack

---

```
assign up_ack = (  
  up_rack |  
  up_wack  
)
```

ack is ack for both, or them so either may pass

## **s\_wb\_ack**

---

```
assign s_wb_ack = up_ack
```

combined uP ack is wishbone ack.